VIT

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name :- Akshat Agarwal

Registration No. :- 23BKT0003

Faculty :- Naveen Kumar J

Course Name :- Operating System Theory

Course Code :- BCSE303L

Digital Assignment

(Topic : CPU Scheduling Algorithms :

Round Robin Scheduling Algorithm)

Question Statement :

Digital Assessment - 1

- ❑ To design and develop an interactive animation tool that visually demonstrates the working of a specific Operating System (OS) algorithm. This tool will serve as a teaching aid to help students and educators understand how the algorithm works step by step. Choose any one algorithms from the given list:
  - ❑ **CPU Scheduling Algorithms**: FCFS, SJF, Priority Scheduling, Round Robin, hybrid.
  - ❑ **Deadlock Algorithms**: Banker's Algorithm, Deadlock Detection.
  - ❑ **Page Replacement Algorithms**: FIFO, LRU, Optimal Page Replacement.
  - ❑ **Disk Scheduling Algorithms**: FCFS, SSTF, SCAN, C-SCAN.
  - ❑ **Memory Allocation**: First Fit, Best Fit, Worst Fit.

Digital Assessment - 1

Functionality (10)	❑ The tool must accept user input for required parameters (e.g., process burst time, arrival time, priority, page requests, etc.). ❑ It should animate the working of the algorithm step by step, showing states like queues, memory frames, disk head movement, allocation tables, etc. respective to the algorithms ❑ The animation must clearly show transitions and results (e.g., waiting time, turnaround time, page faults, etc.)
User Interface (10)	❑ The tool must have a simple, intuitive interface. ❑ Clear labels and instructions for input and output. ❑ Visuals must be clear and easy to understand.
Document and Demonstration (10)	❑ A short report (2-4 pages) describing: ❑ The algorithm with an explanation of its working. ❑ How the tool is designed and how to use it. ❑ Screenshots of the tool in action. ❑ Challenges faced and how you solved them. ❑ Source Code Complete

Round Robin CPU Scheduling Algorithm

1. Explanation of the Algorithm

Definition:

Round Robin (RR) is a preemptive CPU scheduling algorithm used by operating systems to manage multiple processes.

Each process is assigned a fixed time slot called a time quantum.

Processes are scheduled in a circular queue (FIFO order). If a process doesn’t finish within its time quantum, it is preempted and sent to the end of the queue, and the next process is scheduled.

This repeats until all processes are finished.

Key Properties:

- Equal CPU allocation (fairness)
- Starvation-free
- Simple to implement
- Highly dependent on the chosen time quantum value
- Context switches occur at each quantum expiration

Working Principle:

New processes join the end of the ready queue.

Each process executes for one quantum.

If the burst time is less than or equal to quantum, it completes.

If burst time exceeds quantum, it’s preempted and re-queued.

Repeat until the queue is empty.

Example Workflow:

Suppose three processes with burst times:

P1: 6

P2: 3

P3: 7

Time quantum = 2

Time	Process Executed	State
0–2	P1	P2,P3 wait
2–4	P2	P3,P1 requeued
4–6	P3	P1,P2 wait
...	...	(continue until done)

After each quantum, update the remaining time and queue order. After this We Calculate turnaround/waiting times.

Round Robin Visualizer User Guide

1. **Enter Number of Processes**
   - Input how many processes you want to simulate in the "No. of Processes" field.
   - Click the **Generate** button to create input fields for each process's burst and arrival times.
2. **Input Burst and Arrival Times**
   - For every generated process input row, enter the **burst time** (required CPU time) and **arrival time** (when the process arrives in the system).
   - If no arrival time is given, default is zero.
3. **Set Time Quantum and Speed**
   - Set the **Time Quantum** (time slice each process gets in the CPU). Typical values are 1 or 2.

- Set the **Speed** (milliseconds per unit of execution) to determine animation speed.

4. **Load Processes**
   - Click the **Load** button to initialize the process queue with the entered parameters.
   - This enables the **Step** and **Play** buttons to start simulation.
5. **Step Through Simulation**
   - Click **Step** to execute the simulation one time quantum at a time.
   - Watch processes move through CPU, ready queue update, and the Gantt chart fill up step-by-step.
6. **Play Continuous Simulation**
   - Click **Play** to automatically run the simulation continuously at the set speed.
   - Click **Play (Pause)** again to pause.
7. **Observe Outputs**
   - **Ready Queue** shows processes waiting with their remaining burst times.
   - **CPU panel** shows the current running process and its executing time.
   - **Gantt Chart** visually represents process execution slices with aligned timeline ticks.
   - **Process Table** lists each process's burst, arrival, remaining burst, waiting time, and turnaround time dynamically.
   - **Summary statistics** display the average waiting and turnaround times after simulation completes.
8. **Reset and Restart**

- Click **Reset** at any time to clear all processes and outputs, and start fresh.

2. Tool Design and Usage

User Inputs:

- Number of processes
- Burst times for each process
- Arrival times (for advanced simulation)
- Time quantum

Functionality:

- Add processes and adjust burst/arrival times
- Show processes in ready queue
- Step-by-step animation of CPU execution:
- Highlight active process
- Update/remain times and queues after each quantum
- Represent context switches

Display results:

- Waiting time,
- Turnaround time
- Order of execution (Gantt chart)

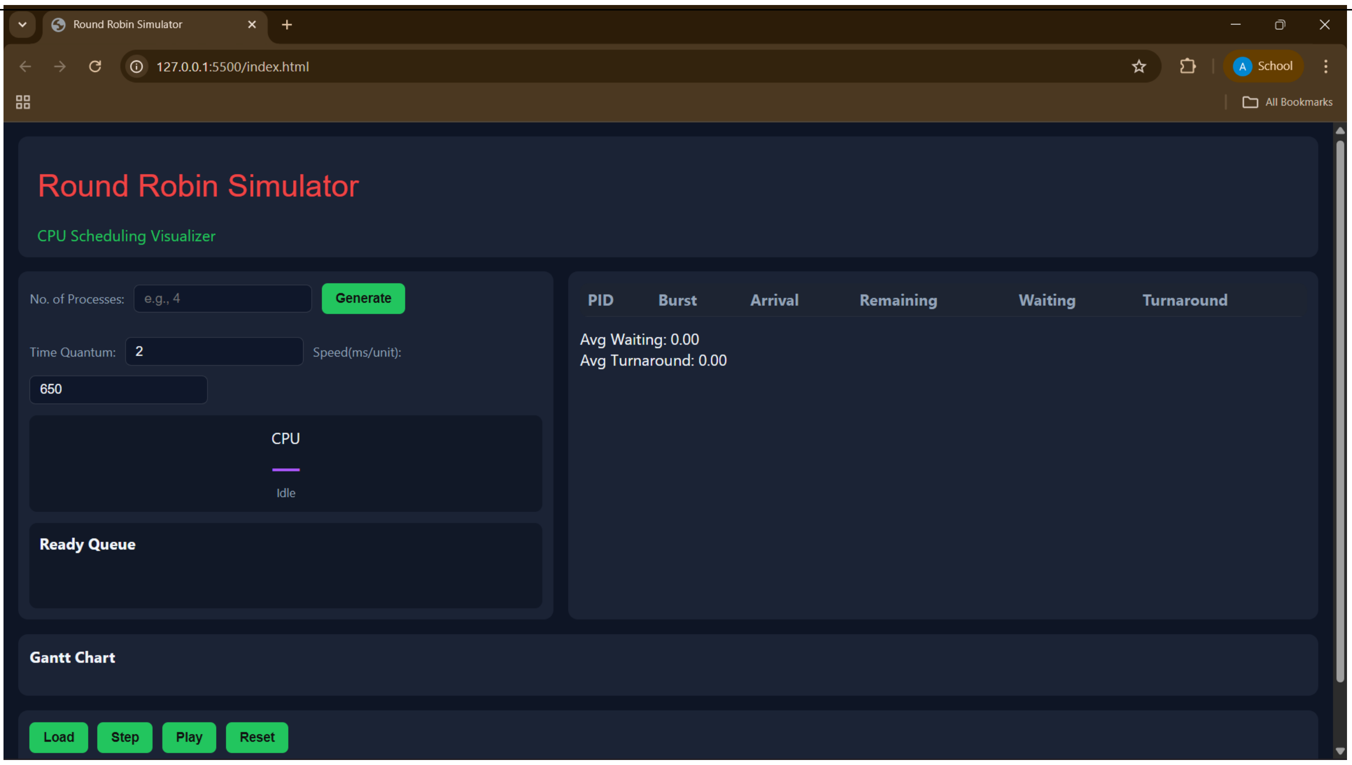
Sample Interface Features:

- Simple input form for process data
- Visual queue to show process order
- Animated Gantt chart of execution timeline
- Labels for all transitions and results

3. Screenshots (conceptual, as per tool in development)

Example Screenshot 1:
Inputs form: Enter burst time, arrival time, quantum time then "Add Process"

Sample Input :



Round Robin Simulator

CPU Scheduling Visualizer

No. of Processes: e.g., 4 Generate

Time Quantum: 2 Speed(ms/unit): 650

CPU

Idle

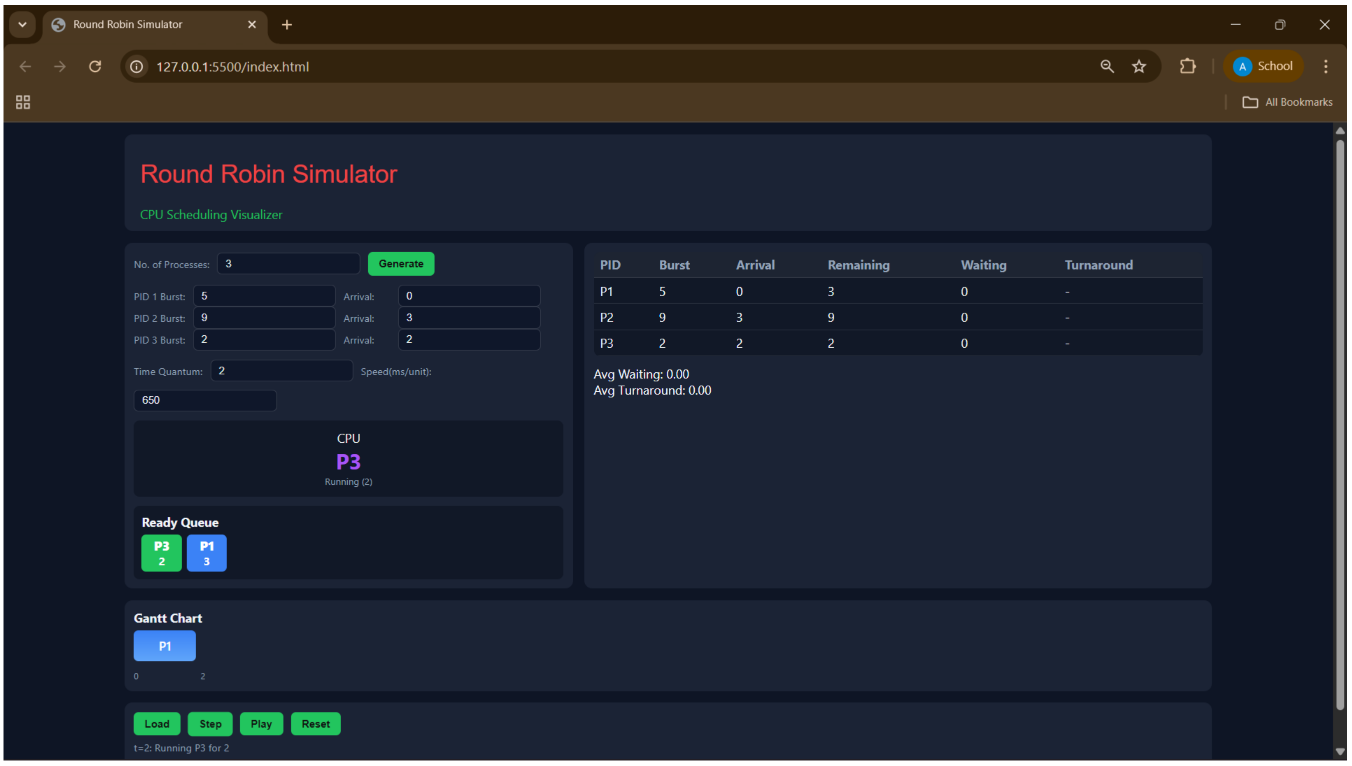
Ready Queue

PID	Burst	Arrival	Remaining	Waiting	Turnaround

Avg Waiting: 0.00
Avg Turnaround: 0.00

Gantt Chart

Load Step Play Reset



Round Robin Simulator

CPU Scheduling Visualizer

No. of Processes: 3 Generate

PID 1 Burst:	5	Arrival:	0
PID 2 Burst:	9	Arrival:	3
PID 3 Burst:	2	Arrival:	2

Time Quantum: 2 Speed(ms/unit): 650

CPU
P3
Running (2)

Ready Queue
P3 2 | P1 3

PID	Burst	Arrival	Remaining	Waiting	Turnaround
P1	5	0	3	0	-
P2	9	3	9	0	-
P3	2	2	2	0	-

Avg Waiting: 0.00
Avg Turnaround: 0.00

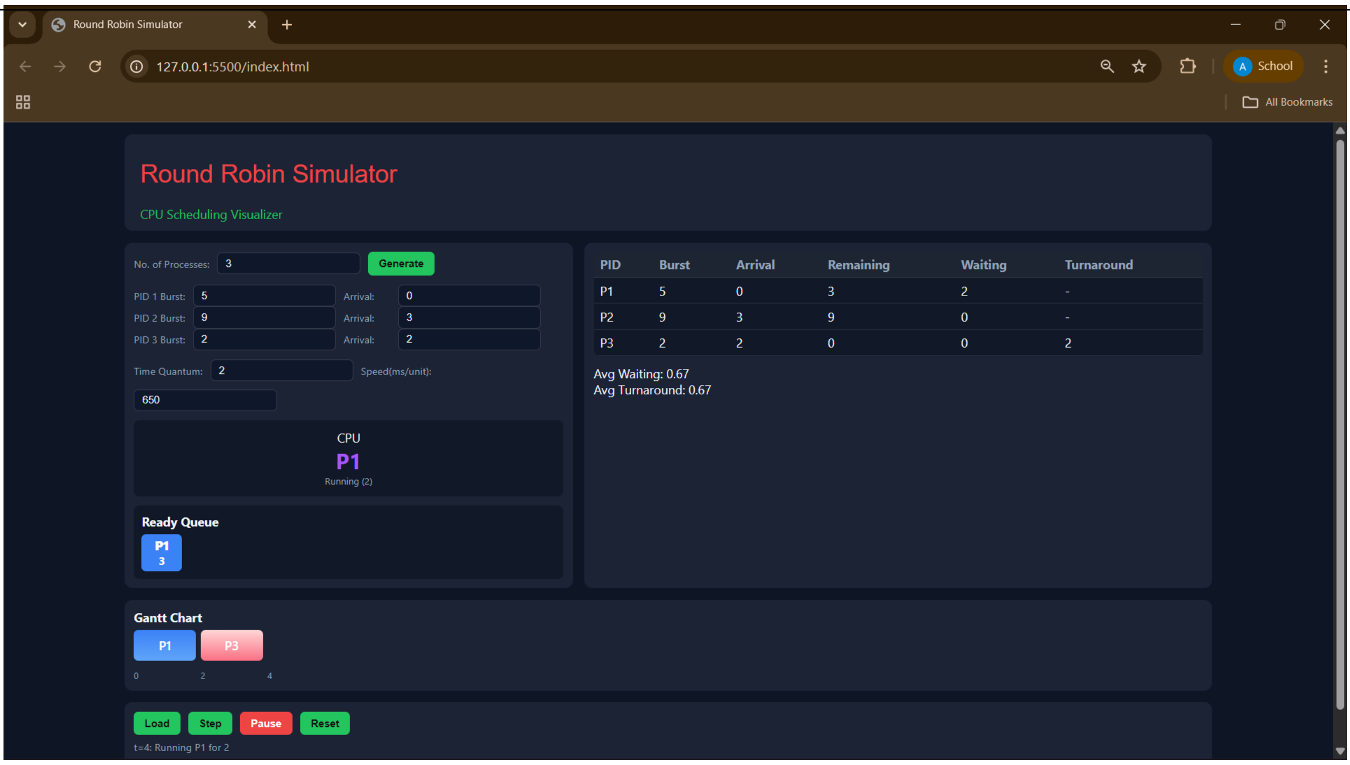
Gantt Chart
P1
0 2

Load Step Play Reset

t=2: Running P3 for 2

Example Screenshot 2:

Ready queue visual: Shows current order and remaining times



Round Robin Simulator

CPU Scheduling Visualizer

No. of Processes: 3 Generate

PID 1 Burst:	5	Arrival:	0
PID 2 Burst:	9	Arrival:	3
PID 3 Burst:	2	Arrival:	2

Time Quantum: 2 Speed(ms/unit): 650

CPU
P1
Running (2)

Ready Queue
P1
3

PID	Burst	Arrival	Remaining	Waiting	Turnaround
P1	5	0	3	2	-
P2	9	3	9	0	-
P3	2	2	0	0	2

Avg Waiting: 0.67
Avg Turnaround: 0.67

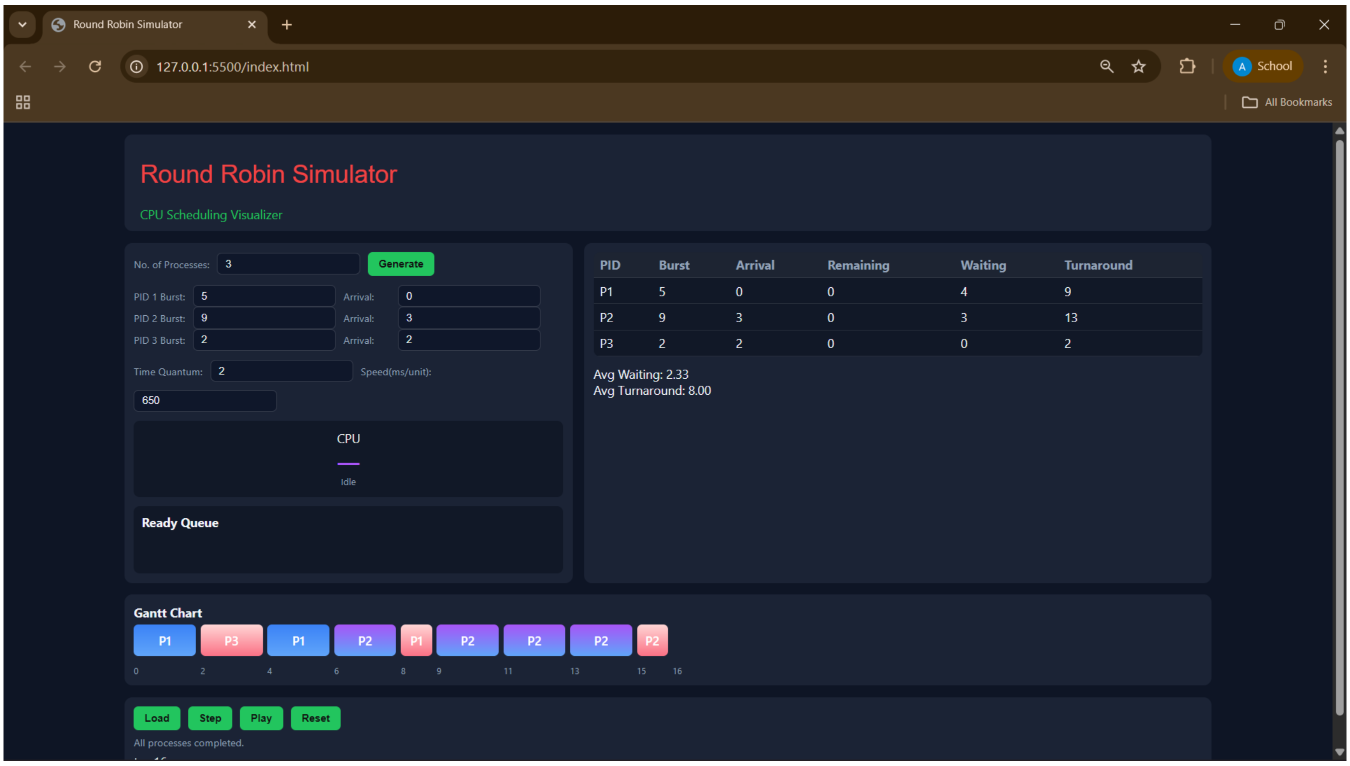
Gantt Chart

Load Step Pause Reset

t=4: Running P1 for 2

Example Screenshot 3:

Animated step: CPU executing a process, requeueing after quantum time



Round Robin Simulator

CPU Scheduling Visualizer

No. of Processes: 3 Generate

PID 1 Burst:	5	Arrival:	0
PID 2 Burst:	9	Arrival:	3
PID 3 Burst:	2	Arrival:	2

Time Quantum: 2 Speed(ms/unit): 650

CPU
Idle

Ready Queue

PID	Burst	Arrival	Remaining	Waiting	Turnaround
P1	5	0	0	4	9
P2	9	3	0	3	13
P3	2	2	0	0	2

Avg Waiting: 2.33
Avg Turnaround: 8.00

Gantt Chart

Load Step Play Reset

All processes completed.

4. Challenges Faced

- Designing a clear and interactive animation that accurately visualizes all queue transitions and context switches.
- Achieving a simple, intuitive user interface with easy-to-follow stepwise execution.
- Correctly computing waiting and turnaround times, especially with different arrival times.
- Communicating results in a visually clear manner (graphical and numerical).
- Ensuring code modularity for easy updates and debugging.

5. Source Code

WebPage Code :

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Round Robin Simulator</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<div class="app-layout">
  <!-- Title Bar -->
  <div class="title-bar">
    <h1 class="stylish-regular">Round Robin Simulator</h1>
    <div class="subtitle">CPU Scheduling Visualizer</div>
  </div>
  <!-- Main Content -->
  <div class="main-grid">
    <!-- Left Pane (Inputs + CPU + Queue) -->
    <div class="left-pane">
      <div class="input-row">
        <label for="numProcesses">No. of Processes:</label>
        <input type="number" id="numProcesses" min="1" placeholder="e.g.,
4">
        <button id="generateBtn">Generate</button>
      </div>
      <div id="processInputs"></div>
      <div class="input-row">
        <label for="timeQuantum">Time Quantum:</label>
        <input type="number" id="timeQuantum" value="2" min="1">
        <label for="speed">Speed(ms/unit):</label>
        <input type="number" id="speed" value="650" min="100">
      </div>
      <!-- CPU & Ready Queue -->
      <div class="cpu">
        <div class="label">CPU</div>
        <div id="cpuPid" class="pid">—</div>
        <div id="cpuSub" class="sub">Idle</div>
      </div>
      <div class="queue-area">
        <div class="queue-header"><strong>Ready Queue</strong></div>
        <div id="readyQueue" class="queue"></div>
      </div>
    </div>
    <!-- Right Pane (Table) -->
    <div class="right-pane">
      <table>
        <thead>
          <tr>
            <th>PID</th>
            <th>Burst</th>
            <th>Arrival</th>
            <th>Remaining</th>
            <th>Waiting</th>
            <th>Turnaround</th>
          </tr>
        </thead>
```

```html
        <tbody id="statsBody"></tbody>
      </table>
      <div class="summary">
        <div>Avg Waiting: <span id="avgWaiting">0.00</span></div>
        <div>Avg Turnaround: <span id="avgTurn">0.00</span></div>
      </div>
    </div>
  </div>
  <!-- Gantt Chart -->
  <div class="gantt-area">
    <div class="gantt-header"><strong>Gantt Chart</strong></div>
    <div id="ganttTrack" class="gantt"></div>
    <div id="ganttTimeline" class="gantt-timeline"></div>
  </div>
  <!-- Control Pane -->
  <div class="control-pane">
    <div class="btn-row">
      <button id="loadBtn" disabled>Load</button>
      <button id="stepBtn" disabled>Step</button>
      <button id="playBtn">Play</button>
      <button id="resetBtn">Reset</button>
    </div>
    <div class="explain" id="explain">Generate processes to begin.</div>
    <div id="timeNow">t = 0</div>
  </div>
</div>
<script src="script.js"></script>
</body>
</html>
```

CSS

```css
:root {
  --bg: #0e1525;
  --card: #1b2335;
  --muted: #94a3b8;
  --text: #f1f5f9;
  --accent1: #3b82f6;
  --accent2: #a855f7;
  --success: #22c55e;
  --danger: #ef4444;
  --btn-default: #22c55e;
  --btn-active: #ef4444;
  font-family: 'Inter', system-ui, sans-serif;
}

body {
  margin: 0;
  background: var(--bg);
  color: var(--text);
  padding: 15px;
}

.stylish-regular { font-family: 'Stylish', sans-serif; font-weight: 400;
font-size: 2rem; color: #ef4444; }
.subtitle { color: #22c55e; }

.app-layout { display: flex; flex-direction: column; gap: 15px; max-width:
1400px; margin: 0 auto; }

.title-bar { background: var(--card); padding: 12px 20px; border-radius:
10px; }

.main-grid { display: grid; grid-template-columns: 1fr 1.4fr; gap: 15px; }

.left-pane { display: flex; flex-direction: column; gap: 12px; background:
var(--card); padding: 12px; border-radius: 10px; overflow-y: auto; }
```

```css
.right-pane { display: flex; flex-direction: column; gap: 12px; background:
var(--card); padding: 12px; border-radius: 10px; overflow-y: auto; }

.input-row, .btn-row { display: flex; flex-wrap: wrap; gap: 10px; align-
items: center; }
.input-row label { min-width: 60px; color: var(--muted); font-size: 13px; }
input { background: #0f172a; border: 1px solid rgba(255,255,255,0.15);
border-radius: 6px; padding: 6px 10px; color: var(--text); }
button { background: var(--btn-default); border: none; padding: 8px 14px;
border-radius: 6px; font-weight: 700; cursor: pointer; color: #111;
transition: 0.2s; }
button:hover { transform: scale(1.02); }
button.active { background: var(--btn-active); color: #fff; }

.cpu { display: flex; flex-direction: column; align-items: center; justify-
content: center; background: #111827; border-radius: 8px; padding: 12px; }
.cpu .pid { font-size: 28px; font-weight: 700; color: var(--accent2); }
.cpu .sub { font-size: 12px; color: var(--muted); }

.queue-area { display: flex; flex-direction: column; gap: 6px; background:
#111827; border-radius: 8px; padding: 10px; }
.queue { display: flex; flex-wrap: wrap; gap: 6px; min-height: 40px; }
.proc { background: var(--accent1); padding: 4px 6px; border-radius: 6px;
font-weight: 700; min-width: 40px; text-align: center; color: #fff; }

table { width: 100%; border-collapse: collapse; background: #111827; border-
radius: 8px; overflow: hidden; }
th, td { padding: 6px 8px; text-align: left; border-bottom: 1px solid
rgba(255,255,255,0.1); }
th { color: var(--muted); background: rgba(255,255,255,0.05); }

.gantt-area { display: flex; flex-direction: column; gap: 6px; background:
var(--card); padding: 12px; border-radius: 10px; overflow-x: auto; }
.gantt { display: flex; gap: 6px; overflow-x: auto; padding-bottom: 6px; }
```

```
.gantt-block { min-width: 40px; height: 40px; display: flex; align-items:
center; justify-content: center; color: #fff; font-weight: 600; border-
radius: 6px; flex-shrink: 0; }
.gantt-timeline { display: flex; gap: 6px; font-size: 11px; color: var(--
muted); }
.gantt-time { width: 40px; text-align: center; flex-shrink: 0; }

.control-pane { display: flex; flex-direction: column; gap: 6px; background:
var(--card); padding: 12px; border-radius: 10px; }
.explain { color: var(--muted); font-size: 13px; min-height: 18px; }

@media (max-width: 900px){
  .main-grid { grid-template-columns: 1fr; }
  .input-row, .btn-row { flex-direction: column; align-items: stretch; }
  input, button { width: 100%; }
}
```

JavaScript

```
window.addEventListener('DOMContentLoaded', () => {
  // DOM references
  const numProcInput = document.getElementById('numProcesses');
  const generateBtn = document.getElementById('generateBtn');
  const processInputsDiv = document.getElementById('processInputs');
  const timeQuantumInput = document.getElementById('timeQuantum');
  const speedInput = document.getElementById('speed');
  const loadBtn = document.getElementById('loadBtn');
  const stepBtn = document.getElementById('stepBtn');
  const playBtn = document.getElementById('playBtn');
  const resetBtn = document.getElementById('resetBtn');

  const cpuPid = document.getElementById('cpuPid');
  const cpuSub = document.getElementById('cpuSub');
  const readyQueueDiv = document.getElementById('readyQueue');
  const statsBody = document.getElementById('statsBody');
  const ganttTrack = document.getElementById('ganttTrack');
  const ganttTimeline = document.getElementById('ganttTimeline');
  const explainDiv = document.getElementById('explain');
  const timeNowDiv = document.getElementById('timeNow');
  const avgWaitingSpan = document.getElementById('avgWaiting');
  const avgTurnSpan = document.getElementById('avgTurn');

  const colors = ['#3b82f6', '#a855f7', '#22c55e', '#ef4444', '#fbbf24', '#14b8a6',
'#ec4899'];
  const genColor = pid => colors[(pid - 1) % colors.length];

  let processes = [];
  let readyQueue = [];
  let currentTime = 0;
  let quantum = 2;
  let speed = 650;
  let running = false;
  let completedCount = 0;
  let timer = null;

  generateBtn.addEventListener('click', () => {
    const n = parseInt(numProcInput.value, 10);
    if (!n || n < 1) {
      alert('Enter valid number of processes!');
      return;
    }
    processInputsDiv.innerHTML = '';
    for (let i = 0; i < n; i++) {
      const div = document.createElement('div');
      div.className = 'input-row';
      div.innerHTML = `
        <label>PID ${i + 1} Burst:</label>
        <input type="number" min="1" class="burst" placeholder="Burst time"
required>
```

```
      <label>Arrival:</label>
      <input type="number" min="0" class="arrival" value="0" required>
    `;
    processInputsDiv.appendChild(div);
  }
  loadBtn.disabled = false;
  stepBtn.disabled = true;
  playBtn.disabled = true;
  explainDiv.textContent = 'Enter burst and arrival values, then click Load.';
});

loadBtn.addEventListener('click', () => {
  if (running) return;
  const burstInputs = document.querySelectorAll('.burst');
  const arrivalInputs = document.querySelectorAll('.arrival');
  processes = [];
  for (let i = 0; i < burstInputs.length; i++) {
    const burst = parseInt(burstInputs[i].value, 10);
    const arrival = parseInt(arrivalInputs[i].value, 10);
    if (isNaN(burst) || burst < 1 || isNaN(arrival) || arrival < 0) {
      alert(`Check process ${i + 1} input!`);
      return;
    }
    processes.push({
      pid: i + 1,
      burst,
      arrival,
      remaining: burst,
      waiting: 0,
      turnaround: 0,
      finished: false,
      enqueued: false
    });
  }
  quantum = Math.max(1, parseInt(timeQuantumInput.value, 10) || 2);
```

```
    speed = Math.max(100, parseInt(speedInput.value, 10) || 650);

    resetSimulationState();

    stepBtn.disabled = false;

    playBtn.disabled = false;

    explainDiv.textContent = 'Processes loaded. Step or Play to run simulation.';

  });


  resetBtn.addEventListener('click', () => {

    if (confirm('Reset and clear simulation?')) {

      stopSimulation();

      resetAll();

    }

  });


  stepBtn.addEventListener('click', async () => {

    if (!running) await doStep();

  });


  playBtn.addEventListener('click', () => {

    if (!processes.length) return alert('Load processes first.');

    if (running) stopSimulation();

    else runSimulation();

  });


  function toggleControls(enable) {

    [numProcInput, generateBtn, loadBtn, stepBtn, playBtn, timeQuantumInput,
speedInput].forEach(e => e.disabled = !enable);

  }


  function resetSimulationState() {

    currentTime = 0;

    completedCount = 0;

    readyQueue = [];

    processes.forEach(p => {

      p.remaining = p.burst;
      p.waiting = 0;
      p.turnaround = 0;
      p.finished = false;
      p.enqueued = false;
    });
    ganttTrack.innerHTML = '';
    ganttTimeline.innerHTML = '';
    cpuPid.textContent = '—';
    cpuSub.textContent = 'Idle';
    readyQueueDiv.innerHTML = '';
    updateTime();
    renderStatsTable();
  }

  function resetAll() {
    processes = [];
    readyQueue = [];
    currentTime = 0;
    completedCount = 0;
    running = false;
    timer && clearTimeout(timer);
    timer = null;
    ganttTrack.innerHTML = '';
    ganttTimeline.innerHTML = '';
    cpuPid.textContent = '—';
    cpuSub.textContent = 'Idle';
    readyQueueDiv.innerHTML = '';
    statsBody.innerHTML = '';
    avgWaitingSpan.textContent = '0.00';
    avgTurnSpan.textContent = '0.00';
    updateTime();
    stepBtn.disabled = true;
    playBtn.disabled = true;
    loadBtn.disabled = true;
    toggleControls(true);
```

```
    explainDiv.textContent = 'Generate processes to begin.';
  }

  function updateReadyQueue() {
    readyQueueDiv.innerHTML = '';
    readyQueue.forEach(idx => {
      const p = processes[idx];
      const div = document.createElement('div');
      div.className = 'proc';
      div.style.backgroundColor = genColor(p.pid);
      div.innerHTML = `<div><strong>P${p.pid}</strong></div><div style="font-
size:0.9em;">${p.remaining}</div>`;
      readyQueueDiv.appendChild(div);
    });
  }

  function updateTime() {
    timeNowDiv.textContent = `t = ${currentTime}`;
  }

  function renderStatsTable() {
    statsBody.innerHTML = '';
    let totalWait = 0, totalTurn = 0;
    processes.forEach(p => {
      const tr = document.createElement('tr');
      tr.innerHTML = `
        <td>P${p.pid}</td>
        <td>${p.burst}</td>
        <td>${p.arrival}</td>
        <td>${p.remaining}</td>
        <td>${p.waiting}</td>
        <td>${p.finished ? p.turnaround : '-'}</td>
      `;
      statsBody.appendChild(tr);
      totalWait += p.waiting;
      totalTurn += p.finished ? p.turnaround : 0;
    });
    avgWaitingSpan.textContent = (totalWait / processes.length).toFixed(2);
    avgTurnSpan.textContent = (totalTurn / processes.length).toFixed(2);
  }

  function addGanttBlock(pid, len, startTime, isFinished) {
    const block = document.createElement('div');
    block.className = 'gantt-block';
    block.textContent = `P${pid}`;
    block.style.minWidth = `${40 * len}px`;
    block.style.background = isFinished ? 'linear-gradient(180deg,#ffd6d6,#fb7185)'
: `linear-gradient(180deg, ${genColor(pid)}, #60a5fa)`;
    ganttTrack.appendChild(block);
  }

  // == MINIMAL TIMELINE FIX: Replace this function only ==
  function updateGanttTimeline() {
    ganttTimeline.innerHTML = '';
    let current = 0;
    Array.from(ganttTrack.children).forEach(block => {
      const blockWidth = parseInt(block.style.minWidth, 10) || 40;
      const duration = blockWidth / 40;
      const label = document.createElement('div');
      label.className = 'gantt-time';
      label.style.width = block.style.minWidth;
      label.style.display = 'inline-block';
      label.style.textAlign = 'left';
      label.textContent = current;
      ganttTimeline.appendChild(label);
      current += duration;
    });
    // Add final tick for simulation end
    const endLab = document.createElement('div');
    endLab.className = 'gantt-time';
    endLab.style.width = '40px';
    endLab.style.display = 'inline-block';
    endLab.style.textAlign = 'left';
    endLab.textContent = current;
    ganttTimeline.appendChild(endLab);
  }
  // == End MINIMAL TIMELINE FIX ==

  async function runExecution(idx, quantumTime, startTime) {
    const p = processes[idx];
    cpuPid.textContent = `P${p.pid}`;
    cpuSub.textContent = `Running (${quantumTime})`;
    await new Promise(r => setTimeout(r, speed * quantumTime));
    cpuPid.textContent = '—';
    cpuSub.textContent = 'Idle';
    addGanttBlock(p.pid, quantumTime, startTime, p.remaining - quantumTime <= 0);
    updateGanttTimeline();
  }

  function enqueueArrivals() {
    processes.forEach((p, idx) => {
      if (!p.enqueued && !p.finished && p.arrival <= currentTime) {
        p.enqueued = true;
        readyQueue.push(idx);
      }
    });
  }

  async function doStep() {
    enqueueArrivals();

    if (readyQueue.length === 0) {
      const future = processes.filter(p => !p.finished && !p.enqueued).map(p =>
p.arrival);
      if (future.length === 0) {
        explainDiv.textContent = 'All processes completed.';
        updateReadyQueue();
        renderStatsTable();
        updateTime();
        computeAverages();
        return;
      }
      currentTime = Math.min(...future);
      enqueueArrivals();
      updateReadyQueue();
      updateTime();
      explainDiv.textContent = `Time advanced to ${currentTime}`;
      return;
    }

    const idx = readyQueue.shift();
    const p = processes[idx];
    if (p.finished) return;

    const quantumUsed = Math.min(quantum, p.remaining);
    readyQueue.forEach(i => processes[i].waiting += quantumUsed);

    explainDiv.textContent = `t=${currentTime}: Running P${p.pid} for
${quantumUsed}`;
    await runExecution(idx, quantumUsed, currentTime);

    currentTime += quantumUsed;
    p.remaining -= quantumUsed;

    if (p.remaining <= 0) {
      p.finished = true;
      p.turnaround = currentTime - p.arrival;
      completedCount++;
      explainDiv.textContent = `P${p.pid} finished at t=${currentTime}`;
    } else {
      enqueueArrivals();
      readyQueue.push(idx);
      explainDiv.textContent = `P${p.pid} preempted; remaining ${p.remaining}`;
    }

    updateReadyQueue();
    renderStatsTable();
    updateTime();

    if (completedCount === processes.length) {
      explainDiv.textContent = 'All processes completed.';
      computeAverages();
      stopSimulation();
    }
  }

  async function runSimulation() {
    running = true;
    toggleControls(false);
    playBtn.textContent = 'Pause';
    playBtn.classList.add('active');
    while (running && completedCount < processes.length) {
      await doStep();
      await new Promise(r => setTimeout(r, 100));
    }
    toggleControls(true);
    running = false;
    playBtn.textContent = 'Play';
    playBtn.classList.remove('active');
  }

  function stopSimulation() {
    running = false;
    toggleControls(true);
    playBtn.textContent = 'Play';
    playBtn.classList.remove('active');

    timer && clearTimeout(timer);

    timer = null;

  }


  function computeAverages() {

    const totalWait = processes.reduce((a, p) => a + p.waiting, 0);

    const totalTurn = processes.reduce((a, p) => a + (p.turnaround || 0), 0);

    avgWaitingSpan.textContent = (totalWait / processes.length).toFixed(2);

    avgTurnSpan.textContent = (totalTurn / processes.length).toFixed(2);

  }


  function init() {

    resetAll();

  }


  init();

});
```

Python Code / Terminal Interpretation :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 100

typedef struct {
    int pid;
    int burst;
    int remaining;
    int waiting;
    int turnaround;
} Process;

Process proc[MAX];
int n, tq;

/* Circular queue implementation */
int q[MAX];
int qfront = 0, qrear = 0;
```

```c
void enqueue(int x) {
    int next = (qrear + 1) % MAX;
    if (next == qfront) {
        printf("Queue overflow!\n");
        exit(1);
    }
    q[qrear] = x;
    qrear = next;
}

int dequeue() {
    if (qfront == qrear) return -1; // empty
    int val = q[qfront];
    qfront = (qfront + 1) % MAX;
    return val;
}

bool qempty() {
    return qfront == qrear;
}

void printReadyQueue() {
    if (qempty()) {
        printf("Empty");
        return;
    }
    int i = qfront;
    bool first = true;
    while (i != qrear) {
        if (!first) printf(" ");
        printf("P%d", proc[q[i]].pid);
        first = false;
        i = (i + 1) % MAX;
    }
}

/* Input */
void inputProcesses() {
    printf("\nEnter number of processes: ");
    if (scanf("%d", &n) != 1) exit(0);
    printf("Enter Time Quantum: ");
    if (scanf("%d", &tq) != 1) exit(0);

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time of Process P%d: ", proc[i].pid);
        if (scanf("%d", &proc[i].burst) != 1) exit(0);
        proc[i].remaining = proc[i].burst;
        proc[i].waiting = 0;
        proc[i].turnaround = 0;
    }
}

/* Round Robin: prints step-by-step Gantt chart + ready queue */
void roundRobin() {
    int time = 0, completed = 0;

    /* reset queue */
    qfront = qrear = 0;

    /* initially enqueue all processes (arrival = 0 assumption) */
    for (int i = 0; i < n; i++) enqueue(i);

    printf("\n--- Step by Step Gantt Chart ---\n");
    printf("Interval\t| Executed Process | Ready Queue (during execution)\n");
    printf("-------------------------------------------------------------\n");

    while (completed < n) {
        int idx = dequeue();
        if (idx == -1) { /* shouldn't happen if completed < n, but safeguard */
            printf("No process in ready queue but not all completed. Exiting.\n");
            break;
        }

        if (proc[idx].remaining <= 0) {
            /* If this process had 0 remaining (edge), skip */
            continue;
        }

        int exec = (proc[idx].remaining > tq) ? tq : proc[idx].remaining;
        int start = time;
        time += exec;
        proc[idx].remaining -= exec;

        /* Print this slice and the ready queue (which currently contains other waiting
processes) */
        printf("%3d - %3d\t|   P%-3d           | ", start, time, proc[idx].pid);
        printReadyQueue();
        printf("\n");

        if (proc[idx].remaining > 0) {
            /* not finished -> re-enqueue at rear */
            enqueue(idx);
        } else {
            /* finished */
            completed++;
            proc[idx].turnaround = time;                 // arrival=0 => turnaround =
finish time
            proc[idx].waiting = proc[idx].turnaround - proc[idx].burst;
        }
    }
}

/* Final results */
void displayResults() {
    float totalWT = 0, totalTAT = 0;

    printf("\n--- Final Results ---\n");
    printf("PID\tBurst\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++) {
```

```c
        printf("P%d\t%5d\t%7d\t%10d\n",
               proc[i].pid, proc[i].burst, proc[i].waiting, proc[i].turnaround);
        totalWT += proc[i].waiting;
        totalTAT += proc[i].turnaround;
    }

    printf("\nAverage Waiting Time    = %.2f\n", totalWT / n);
    printf("Average Turnaround Time= %.2f\n", totalTAT / n);
}

int main() {
    int choice;
    while (1) {
        printf("\n====== Round Robin Scheduling ======\n");
        printf("1. Input Processes\n");
        printf("2. Run Round Robin Scheduling\n");
        printf("3. Display Final Results\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        if (scanf("%d", &choice) != 1) break;

        switch (choice) {
            case 1:
                inputProcesses();
                break;
            case 2:
                roundRobin();
                break;
            case 3:
                displayResults();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

Web References

1. Tutorialspoint. Operating System - Round Robin Scheduling. Available at: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm
2. StudyTonight. Round Robin CPU Scheduling. Available at: https://www.studytonight.com/operating-system/round-robin-scheduling

Open-Source Projects / Visualizers

GitHub – CPU Scheduling Visualizer (includes Round Robin).
https://github.com/AkshatVitRepo/CPU-Scheduling-Visualizer-