1. First-Come, First-Served (FCFS)

- Explanation: The process that arrives first gets executed first.
- Characteristics:
  - Simple to implement.
  - Non-preemptive, meaning processes run to completion.
- Example: In a ticket queue, the first person in line is served first, irrespective of how long it takes.
- Drawback: It can lead to the Convoy Effect, where short processes are stuck waiting behind long processes.

2. Shortest Job Next (SJN) / Shortest Job First (SJF)

- Explanation: The process with the shortest burst time is executed first.
- Characteristics:
  - More efficient than FCFS in terms of average waiting time.
  - Non-preemptive, once a job is started, it runs to completion.
- Example: If three tasks have burst times of 2ms, 5ms, and 3ms, the task with 2ms will execute first.
- Drawback: It can lead to starvation if short processes keep arriving.

Code Block for FCFS & SJF

```
import React, { useState } from 'react';

const Scheduler = () => {

  const [processes, setProcesses] = useState([]);

  const [processName, setProcessName] = useState('');

  const [burstTime, setBurstTime] = useState('');

  const [algorithm, setAlgorithm] = useState('FCFS');


  // Add process to the list

  const addProcess = () => {

    if (processName && burstTime) {

      const newProcess = {
```

```
      name: processName,
      burstTime: parseInt(burstTime),
    };
    setProcesses([...processes, newProcess]);
    setProcessName('');
    setBurstTime('');
  }
};

// SJF sorting
const sjfScheduling = (processList) => {
  return processList.sort((a, b) => a.burstTime - b.burstTime);
};

// FCFS (no need to sort)
const fcfsScheduling = (processList) => {
  return processList;
};

// Handler for scheduling
const handleScheduling = () => {
  let scheduledProcesses = [];
  if (algorithm === 'FCFS') {
    scheduledProcesses = fcfsScheduling([...processes]);
  } else if (algorithm === 'SJF') {
    scheduledProcesses = sjfScheduling([...processes]);
  }

  // Cal CT & TAT
  let currentTime = 0;
  scheduledProcesses = scheduledProcesses.map((process, index) => {
    currentTime += process.burstTime;
    const completionTime = currentTime;
    const turnaroundTime = completionTime; // Since arrival time is 0
    return {
      ...process,
      completionTime,
      turnaroundTime,
    };
  });

  return scheduledProcesses;
};

const scheduledProcesses = handleScheduling();

return (
  <div className="container mx-auto p-4">
    <h1 className="text-2xl font-bold mb-4">CPU Scheduling Algorithms</h1>

    <div className="mb-4">
      <input
        type="text"
        placeholder="Process Name"
        value={processName}
        onChange={(e) => setProcessName(e.target.value)}
        className="border p-2 rounded mr-2"
      />
      <input
        type="number"
        placeholder="Burst Time"
        value={burstTime}
```

```
          onChange={(e) => setBurstTime(e.target.value)}
          className="border p-2 rounded mr-2"
        />
        <button onClick={addProcess} className="bg-blue-500 text-white p-2 rounded">
          Add Process
        </button>
      </div>

      <div className="mb-4">
        <label className="mr-2">Select Algorithm:</label>
        <select value={algorithm} onChange={(e) => setAlgorithm(e.target.value)} className="border p-
2 rounded">
          <option value="FCFS">First Come First Serve (FCFS)</option>
          <option value="SJF">Shortest Job First (SJF)</option>
        </select>
      </div>

      <h2 className="text-xl font-semibold mb-2">Scheduled Processes</h2>
      <table className="table-auto border-collapse border border-gray-400">
        <thead>
          <tr>
            <th className="border border-gray-300 p-2">Process Name</th>
            <th className="border border-gray-300 p-2">Burst Time (ms)</th>
            <th className="border border-gray-300 p-2">Completion Time (ms)</th>
            <th className="border border-gray-300 p-2">Turnaround Time (ms)</th>
          </tr>
        </thead>
        <tbody>
          {scheduledProcesses.map((process, index) => (
            <tr key={index}>
              <td className="border border-gray-300 p-2">{process.name}</td>
              <td className="border border-gray-300 p-2">{process.burstTime}</td>
              <td className="border border-gray-300 p-2">{process.completionTime}</td>
              <td className="border border-gray-300 p-2">{process.turnaroundTime}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
   );
  };

  export default Scheduler;
```

3. Priority Scheduling

- Explanation: Processes are assigned a priority, and the CPU is allocated to the highest-priority process.
- Characteristics:
  - Priority is based on factors like process importance or resource requirements.
  - Non-preemptive, so once a process starts, it finishes before the next priority process begins.
- Example: In a hospital, a critical patient (high priority) is treated before non-urgent cases.
- Drawback: It can lead to indefinite blocking or starvation for low-priority processes.

4. Non-preemptive Round Robin (with Time Slicing)

- Explanation: Round-robin scheduling allocates a fixed time slice to processes, but in non-preemptive systems, the slice is only given once, and the process finishes before the next one starts.
- Characteristics:
  - Ensures fair CPU time distribution.
  - Works well for processes of similar size and needs.
- Example: If three tasks need equal resources and take about the same time, each is given a fair share of CPU time.

Round Robin CPU Scheduling Example:

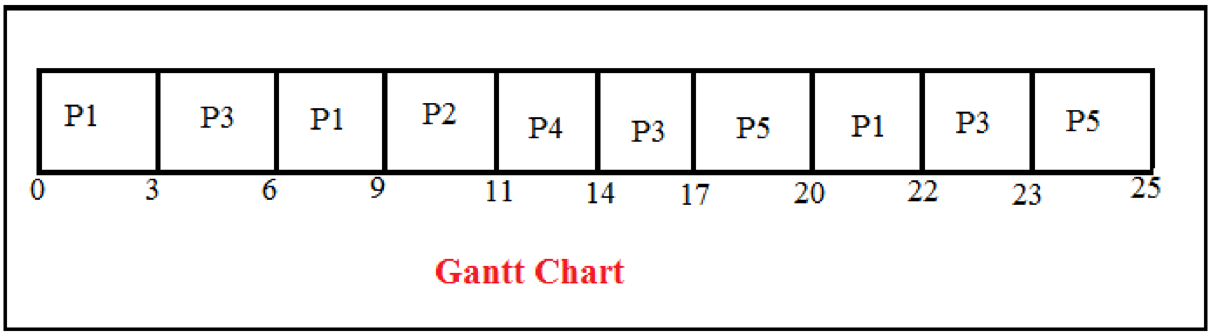
Let's understand the concepts of Round Robin with an example. Suppose we have five processes P1, P2, P3, P4 and P5. The arrival and burst time of each process are mentioned in the following table, as shown below. The time quantum is three units.

Process	**Arrival Time (AT)**	**Burst Time (BT)**
P1	0	8
P2	5	2
P3	1	7
P4	6	3
P5	8	5

Now we have to create the **ready queue** and the **Gantt chart** for Round Robin CPU Scheduler.

Ready queue: P1, P3, P1, P2, P4, P3, P5, P1, P3, P5

Here is the Gantt chart:



Round Robin CPU Scheduling Example:

Let's understand the concepts of Round Robin with an example. Suppose we have five processes P1, P2, P3, P4 and P5. The arrival and burst time of each process are mentioned in the following table, as shown below. The time quantum is three units.

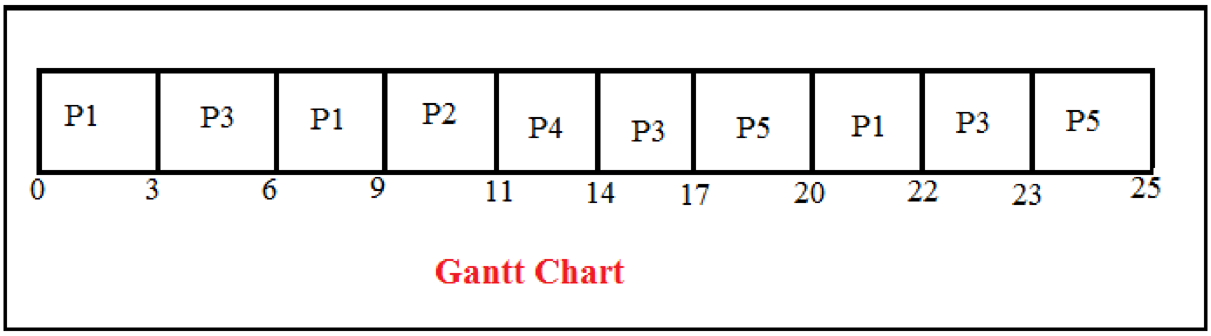
Process	**Arrival Time (AT)**	**Burst Time (BT)**
P1	0	8

P2	5	2
P3	1	7
P4	6	3
P5	8	5

Now we have to create the **ready queue** and the **Gantt chart** for Round Robin CPU Scheduler.

Ready queue: P1, P3, P1, P2, P4, P3, P5, P1, P3, P5

Here is the Gantt chart:



Understanding Race Conditions

What are Race Conditions?

- **Definition:** A race condition occurs when two or more processes access shared resources (like variables, memory, or files) concurrently, and the final outcome depends on the timing or sequence of their execution.

Why Race Conditions Occur?

- Processes or threads access shared resources simultaneously without proper synchronization, leading to inconsistent or undesirable results.
- Common in **multithreading** environments and **critical sections**.

Example of a Race Condition

- **Bank Account Problem:**
  - Two users (processes) try to withdraw money from the same bank account at the same time. Both see the same balance and attempt withdrawals, but the final balance might be incorrect because of the overlap in access.

Example Code in Pseudocode:

pseudocode

Copy code

```
initial_balance = 100
Process 1: Withdraw(50)
Process 2: Withdraw(50)

Final balance: 0 (Expected) vs. 50 (Race Condition)
```

How to Prevent Race Conditions

- **Mutual Exclusion (Mutex):** Ensures only one process can access critical sections at a time.
- **Locks/Semaphores:** Controls access to shared resources.
- **Atomic Operations:** Certain operations are executed fully without interruption.

Real-World Example:

- **Traffic Control:** If two cars (processes) try to cross a single-lane bridge at the same time (shared resource), a crash (race condition) could occur unless there's a traffic light (mutex) controlling access.

Impact of Race Conditions

- **Inconsistent Results:** Leads to unreliable or corrupted data.
- **System Crashes:** In severe cases, a race condition may cause a system to crash or behave unpredictably.

```
import React, { useState, useEffect } from 'react';

const RaceCondition = () => {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    // Simulating thread 1
    const thread1 = setTimeout(() => {
      console.log('Thread 1 started');
      setCounter((prev) => {
        const updatedCounter = prev + 1;
        console.log('Thread 1 updating counter:', updatedCounter);
        return updatedCounter;
      });
    }, 100); // Small delay

    // Simulating thread 2
    const thread2 = setTimeout(() => {
      console.log('Thread 2 started');
      setCounter((prev) => {
        const updatedCounter = prev + 1;
        console.log('Thread 2 updating counter:', updatedCounter);
        return updatedCounter;
      });
    }, 100); // Same delay as thread 1 to cause a race condition

    // Cleanup the timeouts when component unmounts
    return () => {
      clearTimeout(thread1);
      clearTimeout(thread2);
    };
  }, []);

  return (
    <div className="container mx-auto p-4">
      <h1 className="text-2xl font-bold">Race Condition Demo</h1>
      <p className="text-lg">Shared Counter: {counter}</p>
      <p className="text-sm text-red-600">
        Both threads are trying to update the counter at the same time!
      </p>
    </div>
  );
};
```

```
export default RaceCondition;
```

Fixing the RaceCondition

```
import React, { useState, useEffect } from 'react';

const RaceFix = () => {
  const [sharedVariable, setSharedVariable] = useState(0); // Shared variable
  const [thread1Message, setThread1Message] = useState('');
  const [thread2Message, setThread2Message] = useState('');

  // Function to simulate Thread 1 trying to update the shared variable
  const thread1 = () => {
    setTimeout(() => {
      setSharedVariable((prevValue) => {
        const currentValue = prevValue;
        setThread1Message(`Thread 1 reads sharedVariable as: ${currentValue}`);
        const newValue = currentValue + 1;
        setThread1Message(`Thread 1 sets sharedVariable to: ${newValue}`);
        return newValue; // Returns the updated value
      });
    }, Math.random() * 1000); // Random delay
  };

  // Function to simulate Thread 2 trying to update the shared variable
  const thread2 = () => {
    setTimeout(() => {
      setSharedVariable((prevValue) => {
        const currentValue = prevValue;
        setThread2Message(`Thread 2 reads sharedVariable as: ${currentValue}`);
        const newValue = currentValue + 1;
        setThread2Message(`Thread 2 sets sharedVariable to: ${newValue}`);
        return newValue; // Returns the updated value
      });
    }, Math.random() * 1000); // Random delay
  };

  // Start both threads when the component mounts
  useEffect(() => {
    thread1();
    thread2();
  }, []); // Empty dependency array to run only once on mount

  return (
    <div className="container mx-auto p-4">
      <h1 className="text-2xl font-bold mb-4">Race Condition Fixed Demo</h1>
      <p>Shared Variable: {sharedVariable}</p>
      <p className="text-red-500">{thread1Message}</p>
      <p className="text-blue-500">{thread2Message}</p>
    </div>
  );
};

export default RaceFix;
```