Deadlocks

A Course End Project Report on Operating Systems Laboratory

(Course Code - A8510)

Submitted in the Partial Fulfilment of the

Requirements

for the Award of the Degree of

BACHELOR OF TECHNOLOGY

IN

INFORMATION TECHNOLOGY

SubmittedBy

B. CHIRANJEEVI	**24881A1276**
B.SRI MANIHARIKA	**24881A1278**
D.ARCHANA	**24881A1289**

Under the Esteemed Guidance of

Dr. E.Ravi Kumar

Assistant Professor

VARDHAMAN COLLEGE OF ENGINEERING, HYDERABAD

(AUTONOMOUS)

Affiliated to JNTUH, Approved by AICTE, Accredited by NAAC with A++ Grade, ISO 9001:2015 Certified

Kacharam, Shamshabad, Hyderabad – 501218, Telangana, India

November, 2025

VARDHAMAN COLLEGE OF ENGINEERING

(AUTONOMOUS)

Affiliated to **JNTUH**, Approved by **AICTE**, Accredited by **NAAC** with **A++** Grade, **ISO 9001:2015** Certified Kacharam, Shamshabad, Hyderabad – 501218, Telangana, India

DEPARTMENT OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that the Course End Project titled **"Deadlocks"** is carried out by **"B. Chiranjeevi"** "B.Sri Maniharika" and "D.Archana" with Roll Numbers "24881A1276", "24881A1278"and "23881A1289" towards A8512 -Operating Systems Laboratory course in partial fulfilment of the requirements for the award of degree of Bachelor of Tec**hnology** in **Information Technology** during the Academic year 2025-26.

Signature of the Course Faculty

Dr. E .Ravi Kumar, Assistant Professor, IT

Signature of the HOD

Dr.G.Sreenivasulu

HOD, IT

INDEX

Introduction

A **deadlock** is a situation in which two or more processes become permanently blocked, each waiting for a resource that is held by another process. It commonly occurs in operating systems and multithreaded programs where multiple processes share limited resources like memory, CPU, files, or printers. Deadlocks occur when four conditions exist simultaneously mutual exclusion, hold and wait, no preemption, and circular wait.

Once a deadlock occurs, processes cannot proceed, leading to resource wastage and reduced system performance. Deadlocks can cause programs to freeze or crash if not properly handled. To manage deadlocks, systems use various methods such as deadlock prevention, avoidance, detection, and recovery. Understanding deadlocks is important for designing efficient, reliable, and deadlock-free systems in both operating systems and concurrent programming.

Example:

1) **Process** P1 has Resource R1 and needs R2.

2) **Process** P2 has Resource R2 and needs R1.

Now, both are waiting for each other to release the resource this waiting never ends and deadlocks occur.

Deadlocks usually happen in systems where:

1) Resources are shared among multiple processes.
2) Processes hold some resources while requesting others.

Objective of the Project

The primary objective of this project is to study and understand the concept of deadlocks in operating systems. Deadlocks are a critical issue in process management where two or more processes get stuck permanently, each waiting for a resource held by another. This project focuses on understanding how deadlocks occur, their causes, and the methods used to handle them.

1) **To understand the concept of deadlocks:**
The project aims to explain what deadlocks are, how they occur, and why they are important in operating systems and multithreaded programs.

2) **To study the four necessary conditions for deadlock:**
It focuses on analyzing the four main conditions mutual exclusion, hold and wait, non preemption, and circular wait that must exist for a deadlock to occur.

3)**To analyze real-time situations of deadlocks:**
The project demonstrates how deadlocks appear in practical systems, such as printers, databases, or memory-sharing applications, and studies their impact.

4)**To explore methods for handling deadlocks:**
It covers various techniques like prevention, avoidance, detection, and recovery that help control or eliminate deadlocks in systems.

5)**To design and simulate deadlock scenarios:**
The project includes the creation of programs that simulate how deadlocks occur and how they can be resolved effectively.

Problem statement

In modern operating systems and multithreaded environments, multiple processes often compete for limited system resources such as memory, CPU time, files, and input/output devices. When these resources are shared and not managed properly, a situation known as a deadlock may occur, where two or more processes become permanently blocked, each waiting for resources held by the other. This leads to system inefficiency, reduced performance, and, in severe cases, a complete system halt. Detecting and resolving deadlocks is a challenging task because it requires careful monitoring of resource allocation and process states. Therefore, the main problem addressed in this project is to study, identify, and demonstrate how deadlocks occur and to explore effective techniques for preventing, avoiding, and recovering from them, ensuring smooth and reliable system operation**.**

Software and hardware requirements

Software Requirements:

1. **Operating System:** Windows, macOS, or Linux.
2. **Programming Language:** HTML, CSS and Java Script.
3. **Development Environment:** Code editors such as Visual Studio Code (VS Code), Sublime Text, or Atom.
4. **Web Browser:** A modern browser such as Google Chrome, Mozilla Firefox Microsoft Edge, or Safari is required to execute and test.

Hardware Requirements:

1. **Processor:** Intel i3 or higher / AMD Ryzen 3 or higher.
2. **RAM:** Minimum 4 GB (8 GB recommended for smooth GUI development and testing).
3. **Hard Disk:** At least 200 MB of free disk space is required to store project files and related accets.

4.**Monitor:** A standard HD monitor (15 inches or above) is suitable for viewing and testing the interface.

5.**Keyboard:** A standard keyboard is required for coding and debugging.

These requirements ensure that the system can support the development and execution of the project, including Deadlocks and GUI-based visualization.

Project Description

The Deadlocks Project focuses on understanding, simulating, and analyzing the concept of deadlocks in operating systems and concurrent programming. A deadlock occurs when two or more processes or threads are unable to continue execution because each is waiting for a resource held by the other. This leads to a permanent blocking condition where no process can proceed, causing system inefficiency and possible failure.

The main goal of this project is to study how deadlocks occur, identify the necessary conditions for their existence, and demonstrate them through simulation using programming languages such as Python or Java. The project also explores various deadlock handling techniques, including prevention, avoidance, detection, and recovery.

Through this project, users can visually understand how improper resource allocation and lack of synchronization can lead to deadlocks. It provides practical insight into managing shared resources effectively and ensures smoother system performance in multitasking environments. The implementation may include a graphical user interface (GUI) to simulate processes, resources, and waiting conditions, making the concept easier to grasp for learners.

The scope of this project is to study, analyze, and simulate the occurrence and handling of deadlocks in operating systems and concurrent programming environments. The project focuses on understanding how deadlocks happen when multiple processes compete for shared resources and how they affect overall system performance.

Steps/Algorithm/Procedure

The following steps explain the procedure to understand, simulate, and handle deadlocks

1) Start the Simulation: Initialize the system by defining the number of processes and resources available in the system.

2) Resource Allocation: Assign some resources to each process and keep track of which resources are currently allocated and which are free.

3) Process Requests Resource: Each process requests additional resources as required during execution.

4) Check Resource Availability: The system checks if the requested resources are available.

a) If available → allocate them to the process.

b) If not available → the process must wait.

5) Detect Circular Wait: Continuously monitor the resource allocation graph or wait-for graph to check if a circular chain of processes exists (indicating a deadlock).

6)Deadlock Detection: If a circular wait is found, the system declares that a deadlock has occurred between the involved processes.

7) Deadlock Handling: Apply a suitable method to resolve the deadlock:

a) Prevention – Avoid satisfying one of the four necessary conditions.

b) Avoidance – Use algorithms like the Banker's Algorithm to maintain a safe state.

c)Detection and Recovery – Identify deadlock and release or terminate affected processes.

Source Code

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Deadlock-Free Resource Allocation Demo</title>
   <style>
      body { background: #eef6fa; font-family: Arial, sans-serif;}
      .container { max-width: 950px; margin: 40px auto; background: #fff; padding: 24px; border-
radius: 10px; box-shadow: 0 4px 12px rgba(0,0,0,0.10);}
      h1 { text-align: center; color: #2b6cb0;}
      label { font-weight:bold; color: #2b6cb0;}
      input, button {width: 100%; padding: 9px; margin: 5px 0 9px 0; border-radius:4px;
       font-size:1em; border:1px solid #ccc;}
      button {background: #2b6cb0; color: #fff; border: none; cursor: pointer;}
      button:disabled {background: #888;}
      .output {background: #f3f9fd; margin: 10px 0 20px 0; padding: 15px; border-radius: 5px;}
      table {border-collapse: collapse; margin: 10px 0; width:100%;}
      th, td {border: 1px solid #a6d0fd; padding: 8px 10px; text-align: center;}
      th {background: #d7ecfa;}
      .safe {color: green; font-weight:bold;}
      .execute {color: #004729; font-weight: bold;}
      .anim-box {display: flex; justify-content: space-around; margin-top: 20px;}
      .proc, .res {padding: 20px; border-radius: 10px; text-align: center; transition: 0.8s ease;}
      .proc {background: #f0f7ff; border: 2px solid #2b6cb0; width: 120px;}
      .res {background: #fff8dc; border: 2px solid #da9800; width: 120px;}
      .using {background: #bdf7c0 !important;}
      .released {background: #d1f0ff !important;}
       #warning {color: red; font-weight: bold; margin-top: 5px;}
      .input-error {border: 2px solid red;}
       </style>
        </head>
       <body>
    <div class="container">
   <h1>Deadlock-Free Resource Allocation</h1>

   <label>Available resources (comma separated):</label>
   <input type="text" id="resources" placeholder="e.g. 3,4">
   <div id="warning"></div>

   <div id="processorArea">
      <label>Add Processor Resource Needs:</label>
      <input type="text" id="procResources" placeholder="e.g. 3,2">
      <button onclick="addProcessor()">Add Processor</button>
   </div>

   <div id="allProcs"></div>
   <button onclick="runSimulation()" id="runBtn" disabled>Run Safe Allocation</button>
   <button onclick="resetSimulation()" id="resetBtn" style="display:none;">Reset</button>

   <div class="anim-box" id="visual"></div>
```

```
<div class="output" id="log"></div>
<div id="tableArea"></div>
  </div>

 <script>
let available = [];
let processors = [];

    function addProcessor() {
    const warning = document.getElementById("warning");
    const resourceInput = document.getElementById("resources");
    const procInput = document.getElementById("procResources");
    warning.innerHTML = "";
    resourceInput.classList.remove("input-error");
    procInput.classList.remove("input-error");

    let availableInput = resourceInput.value.trim();
    if (!availableInput) {
        warning.innerHTML = " ⚠️ Please enter available resources first.";
        resourceInput.classList.add("input-error");
        return;
    }

    available = availableInput.split(',').map(Number);
    let input = procInput.value.trim();
    if (!input) return;

    let proc = input.split(',').map(Number);
      // Check mismatch in resource type count
    if (proc.length !== available.length) {
        warning.innerHTML = " ⚠️ Number of resource types must match available resources.";
        procInput.classList.add("input-error");
        return;
    }

    // Check if processor needs exceed available resources
    for (let i = 0; i < proc.length; i++) {
        if (proc[i] > available[i]) {
            warning.innerHTML = ` ⚠️ Processor needs exceed available resources at Resource ${i + 1};
              }
            procInput.classList.add("input-error");
            return;
        }
    }

    // If valid, add processor
    processors.push(proc);
    procInput.value = "";
    updateProcessorList();
```

```
    if (processors.length >= 1) document.getElementById("runBtn").disabled = false;
    }

    function updateProcessorList() {
        let txt = "<b>Added Processors (Needs):</b><ul>";
        processors.forEach((p,i)=>{txt+=`<li>P${i+1}: ${p.join(', ')}</li>`;});
        txt+="</ul>";
        document.getElementById("allProcs").innerHTML = txt;
    }

    function runSimulation() {
        available = document.getElementById('resources').value.split(',').map(Number);
        let log = document.getElementById('log');
        log.innerHTML = `<b>Initial Resources:</b> ${available.join(', ')}<br><hr>`;

        visualizeState("initial");

        // Sequentially execute all processors safely
        let i = 0;
        function executeNext() {
            if (i >= processors.length) {
                buildTable(new Array(processors.length).fill("Executed"));
                return;
            }

            let p = processors[i];
            log.innerHTML += `<b>P${i+1}:</b> <span class='execute'>Accessing
resources...</span><br>`;
            visualizeState(`p${i+1}-using`);
available = available.map((v,k)=>v - p[k]);
            log.innerHTML += `P${i+1} used resources. Remaining available: ${available.join(', ')}<br>`;

            setTimeout(()=>{
                available = available.map((v,k)=>v + p[k]);
                log.innerHTML += `<span class='safe'>P${i+1} released resources.</span> Now available:
${available.join(', ')}<br>`;
                visualizeState(`p${i+1}-released`);
                i++;
                setTimeout(executeNext, 1500);
            }, 1500);
        }

        executeNext();
    }

    function visualizeState(state) {
        const area = document.getElementById("visual");
        let html = "";
```

```
    processors.forEach((_, index)=>{
            html += `<div class="proc" id="p${index+1}">Processor ${index+1}</div>`;
        });
        html += `<div class="res" id="res">Resources</div>`;
     area.innerHTML = html;

        processors.forEach((_, index)=>{
             const procBox = document.getElementById(`p${index+1}`);
            if(state === `p${index+1}-using`) procBox.classList.add("using");
            if(state === `p${index+1}-released`) procBox.classList.add("released");
        });
    }

    function buildTable(statusArr) {
        let table = "<table><tr><th>Processor</th>";
        for(let i=0; i<available.length; i++) table += `<th>Res${i+1}</th>`;
        table += "<th>Status</th></tr>";
        processors.forEach((proc,i)=>{
            let st = "<span class='safe'>Executed</span>";
            table +=
`<tr><td>P${i+1}</td>${proc.map(n=>`<td>${n}</td>`).join('')}<td>${st}</td></tr>`;
        });
        table += "</table>";
        document.getElementById('tableArea').innerHTML = table;
        document.getElementById('runBtn').disabled = true;
        document.getElementById('resetBtn').style.display = 'inline-block';
    }

    function resetSimulation() {
        available = [];
        processors = [];
        document.getElementById("allProcs").innerHTML = "";
        document.getElementById('log').innerHTML = "";
        document.getElementById('tableArea').innerHTML = "";
        document.getElementById('visual').innerHTML = "";
        document.getElementById('warning').innerHTML = "";
        document.getElementById('runBtn').disabled = true;
        document.getElementById('resetBtn').style.display = 'none';
        document.getElementById("resources").classList.remove("input-error");
        document.getElementById("procResources").classList.remove("input-error");
    }
</script>
</body>
</html>
```

Result(s)

Input:

- Available resources: 3,4
- Processor 1 needs: 2,2
- Processor 2 needs: 1,3

Output:

Initial Resources: 3,4

P1: Accessing resources...

P1 used resources. Remaining available: 1,2

P1 released resources. Now available: 3,4

P2: Accessing resources...

P2 used resources. Remaining available: 2,1

P2 released resources. Now available: 3,4

All processors executed safely — No Deadlock detected.

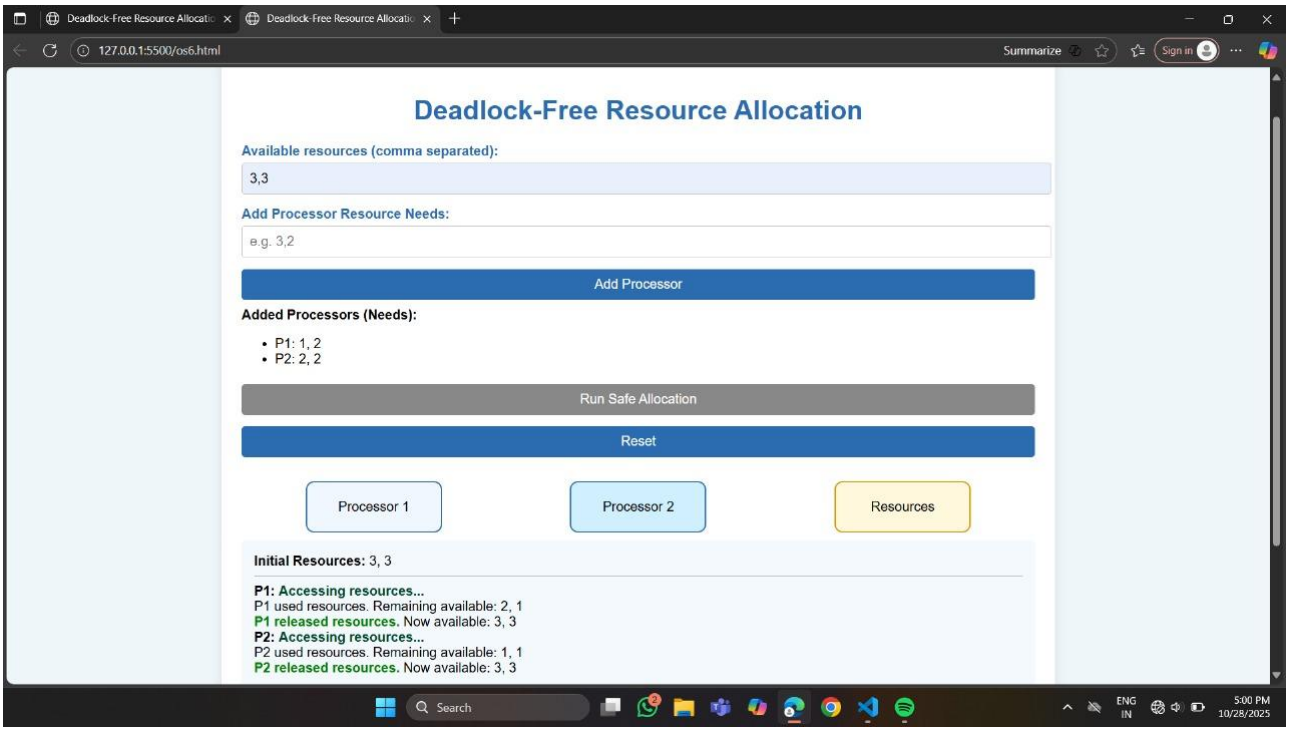


Fig No:1 Allocating Resources to the Processors.

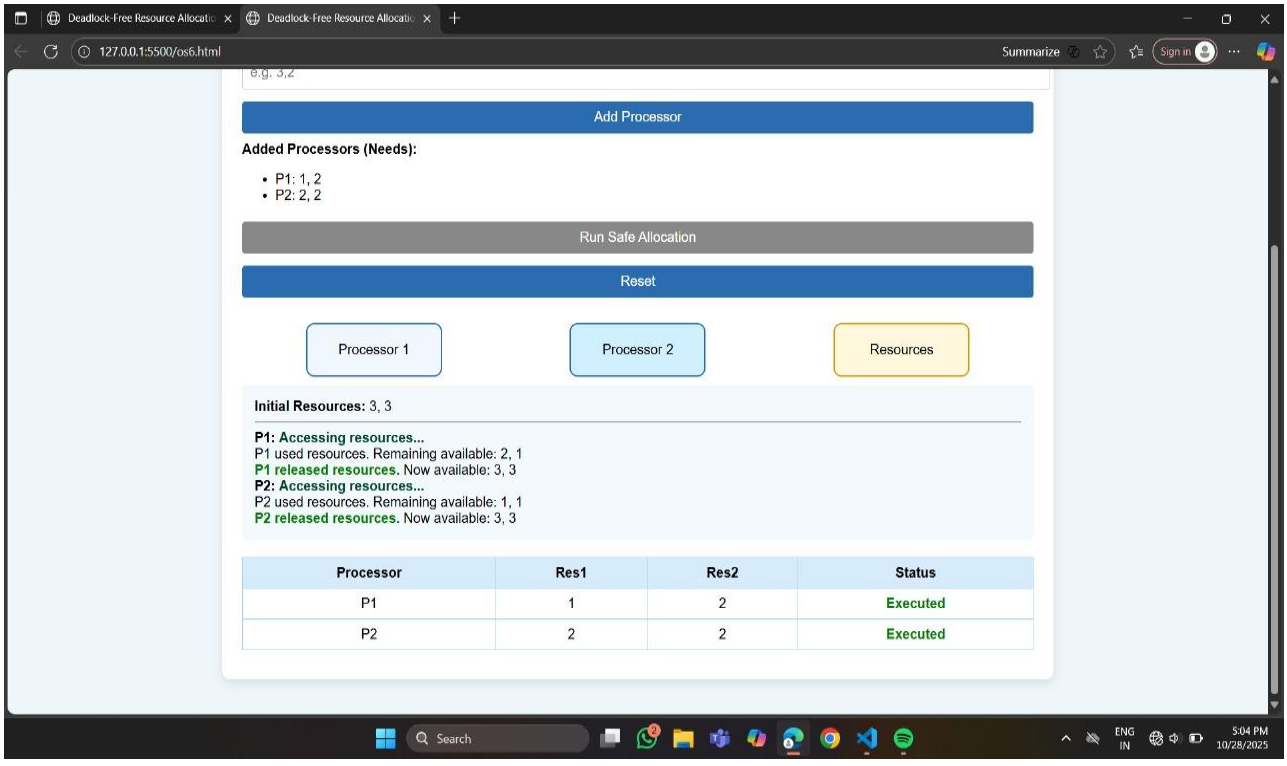


Fig No 2: Showing the Processors are Executed.

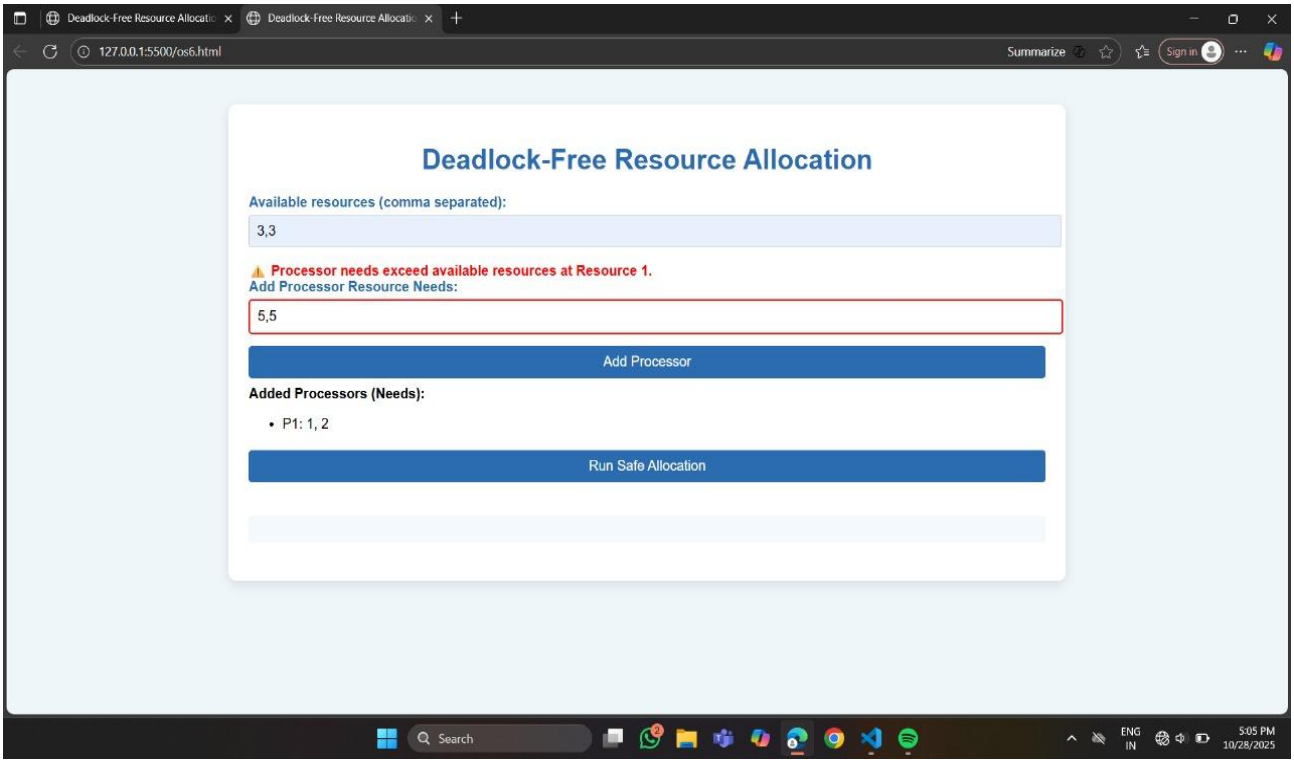


Fig No 3: It shows an error if we allocate exceed Resources.

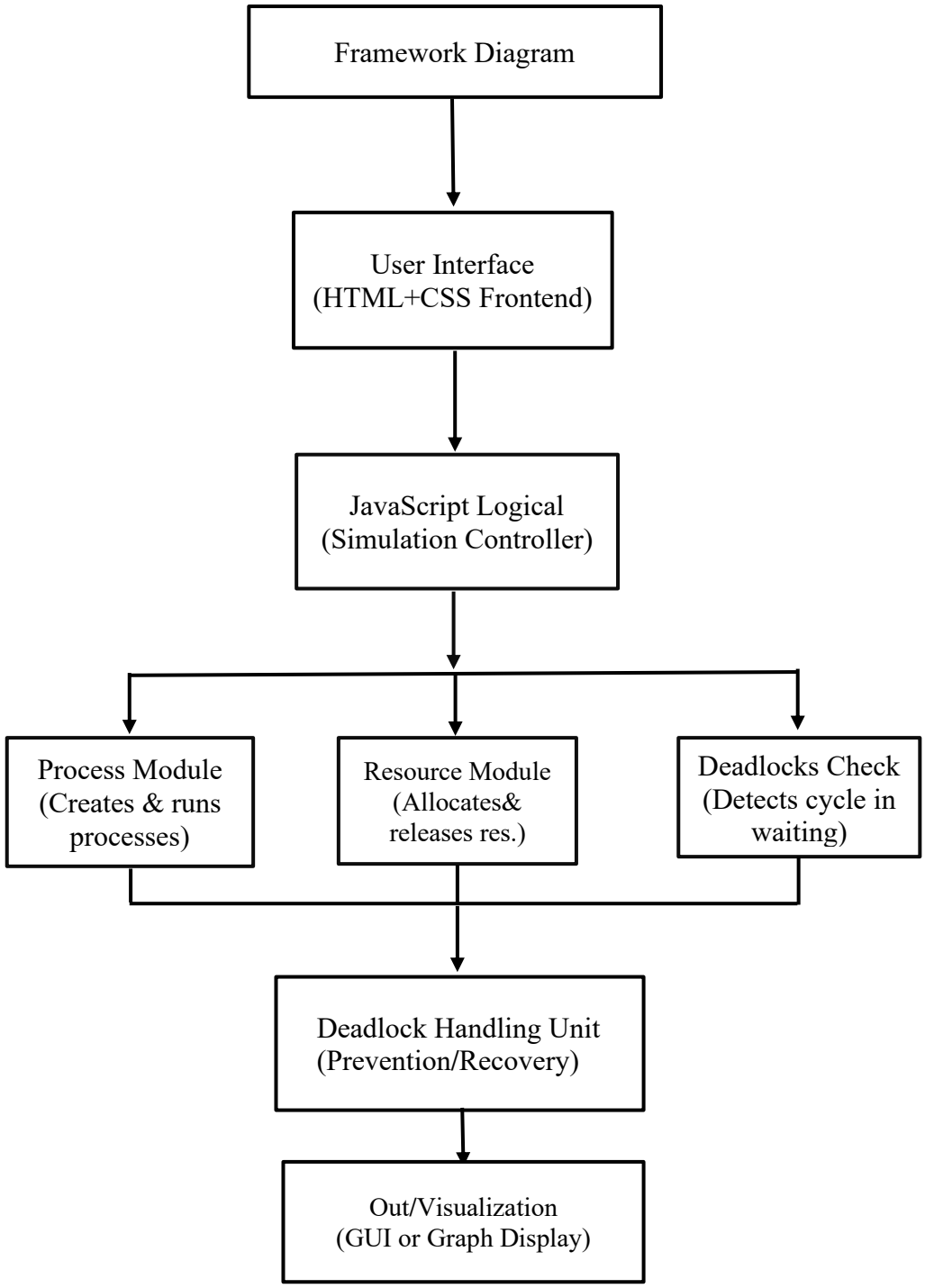
Conclusion and Framework

Conclusion:

The Deadlocks Project successfully demonstrates how deadlocks occur in operating systems when multiple processes compete for shared resources. Through simulation using JavaScript, HTML, and CSS, the project explains the four necessary conditions for deadlock — mutual exclusion, hold and wait, no preemption, and circular wait. It also highlights different techniques to handle deadlocks, such as prevention, avoidance, detection, and recovery. By visualizing these scenarios, the project provides a better understanding of process synchronization, resource allocation, and system reliability. This project helps learners clearly grasp one of the most important concepts in operating systems and concurrent programming

Framework:

- Deadlocks can be handled in four main ways: prevention, avoidance, detection and recovery, and ignorance.
- In prevention, one of the necessary conditions is eliminated to avoid deadlock.
- In avoidance, algorithms like the **Banker's Algorithm** are used to keep the system in a safe state.
- In detection and recovery, deadlocks are allowed to occur but are later detected and resolved by aborting or preempting processes.
- Some systems, such as UNIX, ignore deadlocks as they occur rarely.
- A Resource Allocation Graph (RAG) can be used to represent and detect potential deadlocks.
- Example: If process P1 holds one resource and requests another held by P2, and P2 requests the one held by P1, a circular wait occurs.
- Understanding deadlocks is essential in operating systems, databases, and distributed computing for stable and efficient performance.


Framework Diagram
User Interface
(HTML+CSS Frontend)
JavaScript Logical
(Simulation Controller)
Process Module
(Creates & runs
processes)
Resource Module
(Allocates&
releases res.)
Deadlocks Check
(Detects cycle in
waiting)
Deadlock Handling Unit
(Prevention/Recovery)
Out/Visualization
(GUI or Graph Display)

References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th Edition). Wiley.
2. Stallings, W. (2018). *Operating Systems*: Internals and Design Principles (9th Edition). Pearson Education.
3. Tanenbaum, A. S., & Bos, H. (2015). Modern Operating Systems (4th Edition). Pearson.
4. TutorialsPoint. (n.d.). Deadlocks in Operating System. Retrieved from https://www.tutorialspoint.com
5. GeeksforGeeks. (n.d.). Deadlock in Operating System. Retrieved from https://www.geeksforgeek
6. The YouTube channel for Sudhakar Atchala: https://www.youtube.com/c/SudhakarAtchala