Operating Systems

LAB MANUAL

Project Title: Banker’s Algorithm

Submitted to: Sir Umar Malik

Submitted By: Alina Liaquat

Roll No: BSCSM-A-23-19

Department of Computer Science

UNIVERSITY OF LAYYAH MAIN CAMPUS HAFIZABAD

ACKNOWLEDGEMENT

I extend my sincere gratitude to my course instructor **Sir Umar Malik** for providing me the opportunity to work on the "OS Deadlock Simulator" project. His guidance and expertise in Operating Systems, particularly in process management and deadlock handling, were invaluable throughout this project's development.

His mentorship helped me implement the Banker's Algorithm effectively and create a practical simulation that demonstrates core OS principles. This project has significantly enhanced my understanding of resource management and deadlock prevention strategies in operating systems.

PROBLEM STATEMENT

Background

In multiprogramming operating systems, multiple processes compete for limited resources. When processes request resources in a circular waiting pattern, a **deadlock** occurs, causing system stagnation. The **Banker's Algorithm**, developed by Edsger Dijkstra, is a classic deadlock avoidance algorithm that ensures system safety by carefully allocating resources.

Problem Definition

Design and implement an interactive **Deadlock Simulator** that:

1. Visually demonstrates the Banker's Algorithm for deadlock avoidance
2. Simulates real process behavior with modern application names
3. Provides real-time safety analysis and deadlock detection
4. Offers an intuitive interface for resource request management
5. Shows step-by-step execution of safety algorithms

Objectives

- Implement Banker's Algorithm with proper safety checks
- Create visual representations of allocation matrices
- Simulate deadlock scenarios and their resolutions
- Provide educational insights into OS resource management
- Demonstrate practical application of theoretical concepts

THEORETICAL CONCEPTS

1. Deadlock Conditions (Coffman Conditions)

Four necessary conditions for deadlock:

1. **Mutual Exclusion:** Resources cannot be shared
2. **Hold and Wait:** Processes hold resources while waiting for others
3. **No Preemption:** Resources cannot be forcibly taken
4. **Circular Wait:** Circular chain of processes waiting for resources

2. Banker's Algorithm Components

```
Data Structures:

- Available[m]: Available instances of each resource type

- Max[n×m]: Maximum demand of each process

- Allocation[n×m]: Currently allocated resources

- Need[n×m]: Remaining resource needs (Max - Allocation)
```

3. Safety Algorithm

The algorithm determines if the system is in a safe state by finding a safe sequence where all processes can complete execution without deadlock.

4. Resource-Request Algorithm

Handles resource requests by checking:

1. Request ≤ Need
2. Request ≤ Available
3. Temporary allocation simulation
4. Safety check after allocation

ALGORITHM STEPS

A. Banker's Safety Algorithm

```
Step 1: Initialize

    Work = Available

    Finish[i] = false for all processes i

Step 2: Find a process i such that:

    Finish[i] == false AND

    Need[i] ≤ Work for all resources

Step 3: If such process exists:

    Work = Work + Allocation[i]

    Finish[i] = true
```

```
    Add i to Safe Sequence

    Go to Step 2

Step 4: If Finish[i] == true for all i:

    System is in safe state

    Return safe sequence

Else:

    System is unsafe
```

B. Resource Request Algorithm

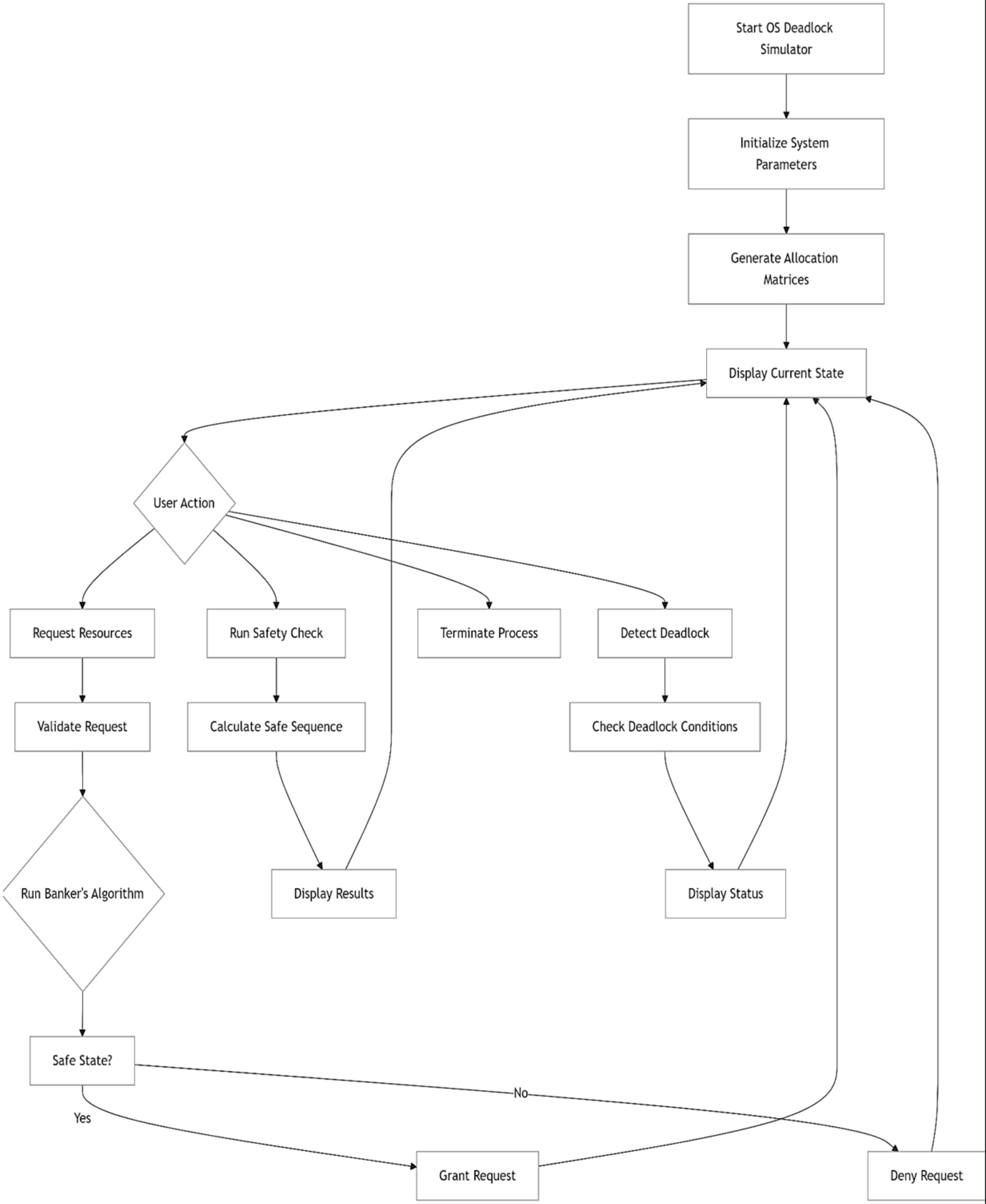
```
Step 1: If Request[i] > Need[i]:

    Error: Process exceeded maximum claim

    Terminate

Step 2: If Request[i] > Available:

    Process must wait (resources not available)

Step 3: Tentatively allocate:

    Available = Available - Request[i]

    Allocation[i] = Allocation[i] + Request[i]

    Need[i] = Need[i] - Request[i]

Step 4: Run Safety Algorithm

    If safe: Allocation made permanent

    If unsafe: Rollback allocation
```

C. Deadlock Detection Algorithm

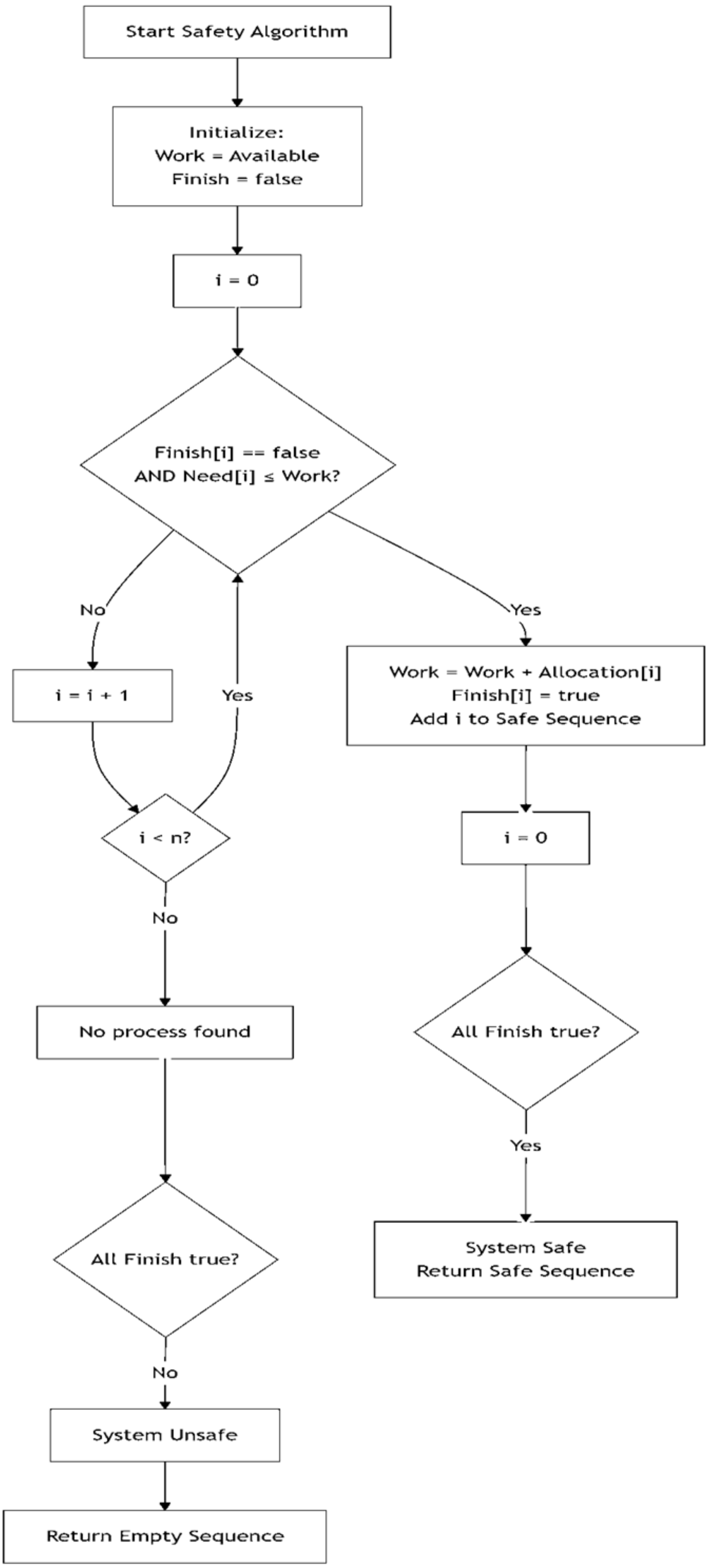
```
Step 1: Initialize

    Work = Available

    Finish[i] = (Allocation[i] == 0) for all i

Step 2: Find process i where:

    Finish[i] == false AND

    Request[i] ≤ Work

Step 3: If found:

    Work = Work + Allocation[i]

    Finish[i] = true

    Repeat Step 2

Step 4: Processes with Finish[i] == false

    are deadlocked
```

FLOWCHARTS

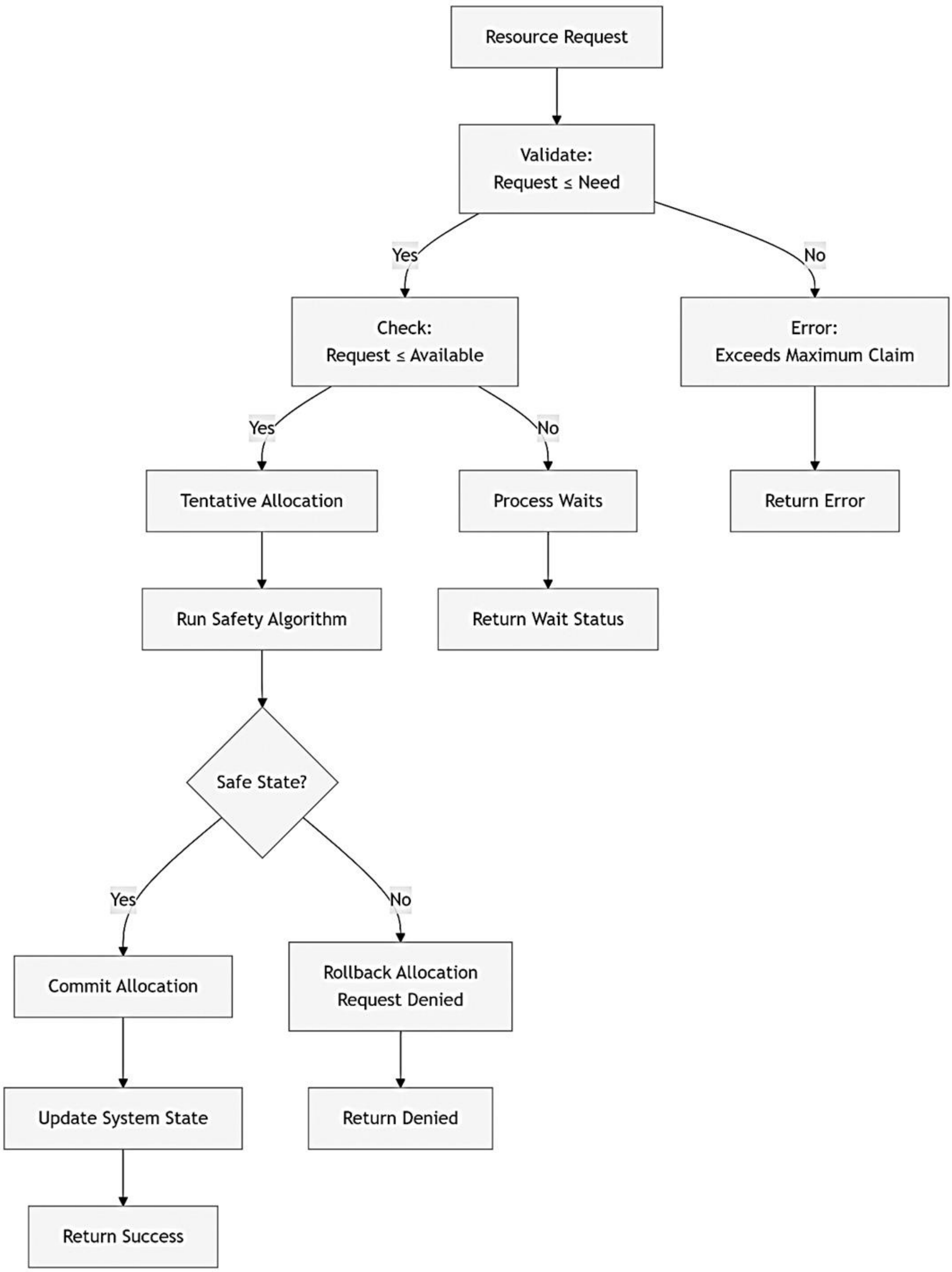
Overall System Flow



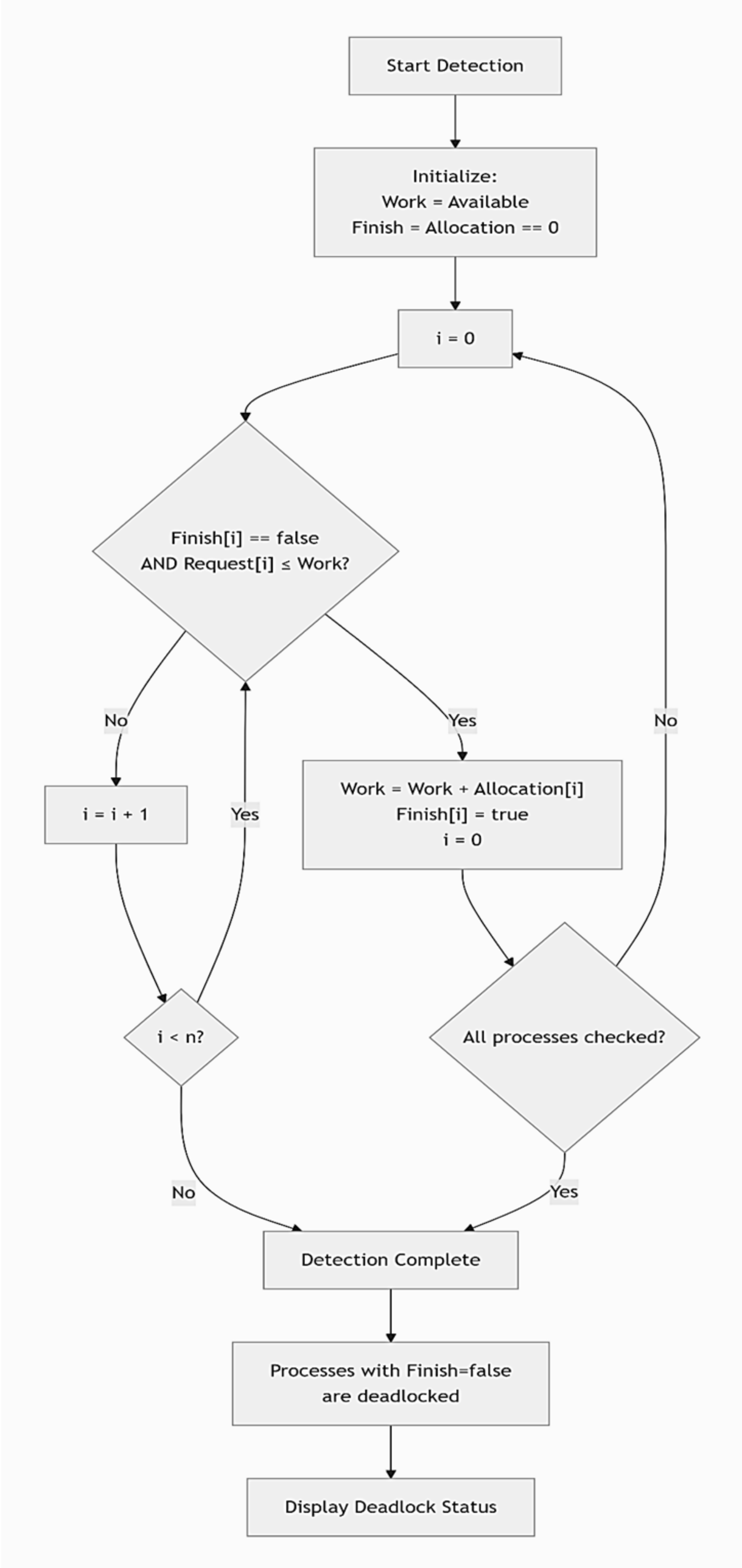
Banker's Safety Algorithm



Resource Request Handling

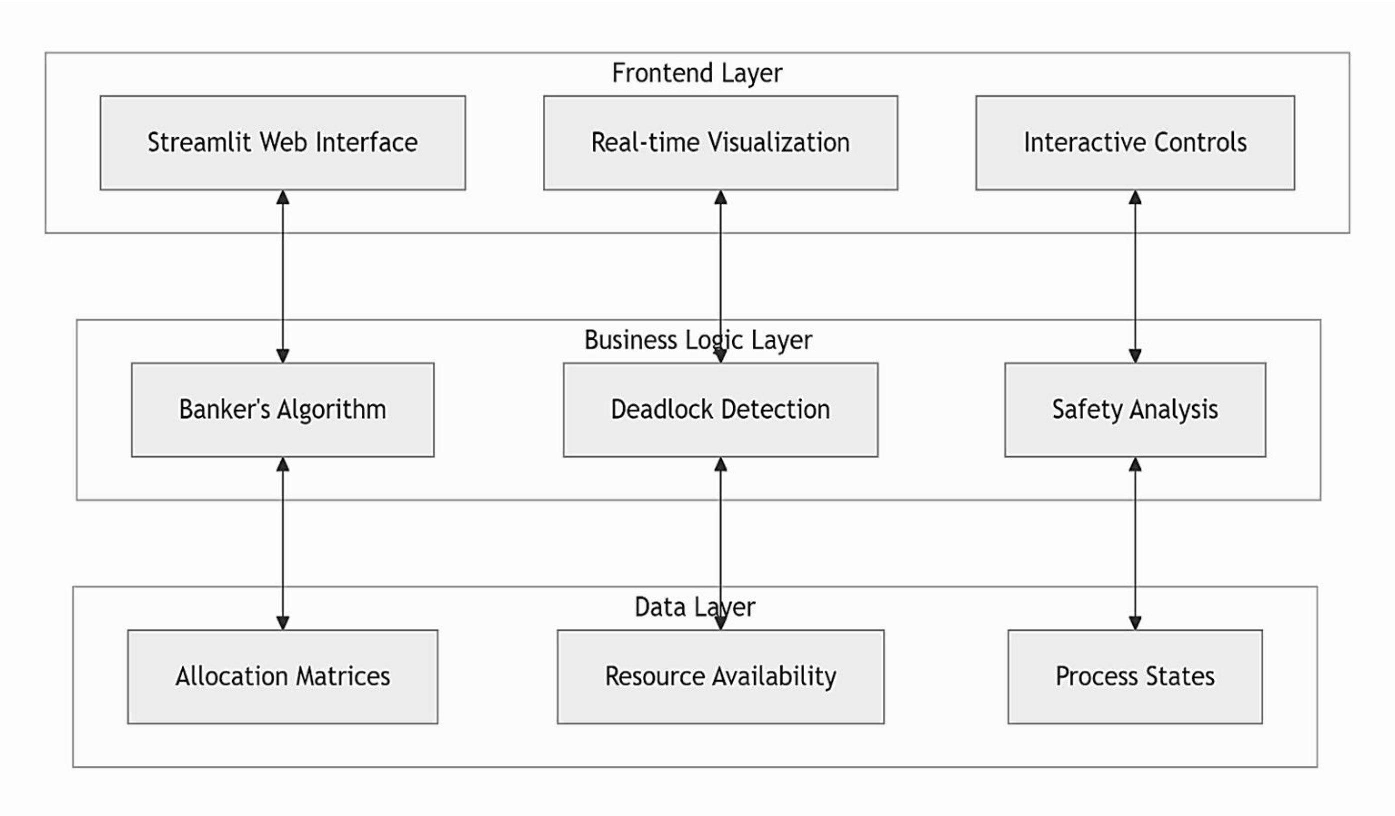


Deadlock Detection Process



IMPLEMENTATION DETAILS

System Architecture



2. Key Features Implemented

- **Dynamic System Initialization:** Configurable processes and resources
- **Real Process Simulation:** Modern application names (Chrome, VS Code, etc.)
- **Visual Matrices:** Real-time display of allocation states
- **Safety Analysis:** Instant safe sequence calculation
- **Deadlock Detection:** Automatic scanning for circular waits
- **Resource Request Management:** Interactive request handling
- **Historical Tracking:** Request history with timestamps

EXPERIMENTS AND OBSERVATIONS

Experiment 1: Safe State Verification

```
Input Configuration:
- Processes: 5
- Resources: 3
- Allocation: Random
- Maximum: Random (higher than allocation)
Observation:
System calculates safe sequence
All processes can complete execution
No deadlock detected
```

Experiment 2: Deadlock Creation

```
Simulation Steps:
1. Allocate all resources
2. Make circular requests
3. Run detection algorithm
Observation:
Deadlock detected
Circular wait identified
Termination option provided
```

Experiment 3: Resource Request Handling

```
Test Cases:

1. Valid request within need → Granted

2. Request exceeds available → Denied

3. Request leads to unsafe state → Denied

Observation:

Banker's algorithm prevents unsafe allocations

System maintains safe state
```

SAMPLE OUTPUTS

Output 1: System Initialization

```
OS Deadlock Simulator Initialized

---------------------------------

Processes: 5

Resources: 3

Allocation Matrix Generated

Available Resources: [4, 3, 2]

System Status: SAFE

Safe Sequence: P1 → P3 → P4 → P2 → P0
```

Output 2: Resource Request

```
Process: Chrome Browser

Request: [1, 0, 1]

Checking Safety...

Request GRANTED

New Safe Sequence: P3 → P1 → P4 → P2 → P0
```

Output 3: Deadlock Detection

```
Deadlock Scan Initiated...

DEADLOCK DETECTED!

Blocked Processes:

- VS Code Editor

- Spotify Player

- Discord Client

Suggestion: Terminate one process
```

CONCLUSION

This lab successfully implemented a comprehensive **Deadlock Simulator** using the Banker's Algorithm. The project demonstrates:

1. **Practical Implementation** of theoretical deadlock concepts
2. **Visual Learning** through interactive matrices and status indicators
3. **Analysis** of system safety and resource allocation
4. **Educational Value** in understanding OS resource management

The simulator effectively bridges the gap between abstract algorithmic concepts and practical system behavior, providing valuable insights into how operating systems prevent and handle deadlocks in real-world scenarios.