

**SAVEETHA SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL**

**SCIENCES**

**CAP STONE PROJECT**

**Name:**

I Guru Hemanth Reddy(192311222)

**Course and course code:** CSA0459 - Operating Systems(OS)

**Topic:** Deadlock Detection and Resolution in a Multi-Process Operating System

**Introduction:**

In multi-process operating systems, deadlock occurs when a set of processes become blocked, each waiting for a resource held by another, causing the system to halt progress. Effective **deadlock detection and resolution** is essential to ensure system reliability and optimal performance. This project focuses on detecting deadlocks and implementing strategies to resolve them, such as process termination, resource preemption, and rollback.

The goal is to develop methods for identifying and resolving deadlocks in real-time operating systems, minimizing system downtime, and ensuring efficient resource management. The project will explore various algorithms for deadlock detection and propose solutions for recovery, contributing to improved system stability.

This project focuses on detecting deadlocks in a multi-process environment and proposing methods for resolving them. It aims to explore various **deadlock detection algorithms** such as Wait-for Graph, and **deadlock resolution techniques** like process termination, resource preemption, and rollback mechanisms.



**Aim:**

The aim of this project is to develop methods for detecting and resolving deadlocks in a multi-process operating system. The project focuses on identifying deadlock situations using algorithms like the Wait-for Graph and resolving them through strategies such as process termination, resource preemption, and rollback, ensuring efficient resource management and system performance. By implementing these techniques, the project seeks to enhance system stability and prevent resource wastage. The ultimate goal is to provide a robust solution that ensures smooth operation in multi-tasking environments.

**Apparatus Required:**

1. **Computer/Workstation** with sufficient processing power for simulation and testing.
2. **Programming Languages** (C, C++, Java, Python) for implementing algorithms.
3. **Operating System** (Linux, Windows) for real-time testing.
4. **Graph Visualization Tools** like Lucidchart or draw.io for creating and visualizing graphs.
5. **Text Editor/IDE** (Visual Studio, Eclipse, PyCharm) for coding and debugging.
6. **Simulator/Modeling Tools** (SimPy for Python) for simulating process and resource management.
7. **Documentation Tools** (Microsoft Word, LaTeX) for writing reports and results.

**Algorithm:**

Deadlock Detection and Resolution

**1. Deadlock Detection Algorithm (using Wait-for Graph)**

**Step 1:** Initialize a **Wait-for Graph (WFG)** where:

* Processes are nodes.
* Directed edges represent waiting relationships (if Process A is waiting for Process B to release a resource, an edge from A to B is created).

**Step 2:** Traverse the Wait-for Graph to check for cycles:

* Perform **Depth-First Search (DFS)** to detect cycles in the graph.
* A cycle indicates that a deadlock exists since each process in the cycle is waiting for a resource held by another in the same cycle.

**Step 3:** If a cycle is detected, **mark it as a deadlock.**

**2. Deadlock Resolution Algorithm**

Once deadlock is detected, you can apply one of the following methods:

**Method 1: Process Termination**

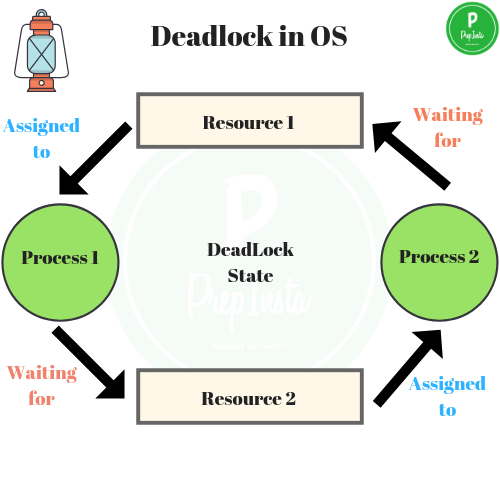
* **Step 1:** Select one or more processes involved in the deadlock.
* **Step 2:** Terminate the selected processes to release their resources.
* **Step 3:** Re-run the detection algorithm to ensure the system is free of deadlocks.

**Method 2: Resource Preemption**

* **Step 1:** Select a process to preempt (forcefully take its resources).
* **Step 2:** Rollback the selected process to a safe state (optional).
* **Step 3:** Reassign the resources to another process.
* **Step 4:** Re-run the detection algorithm to ensure the system is deadlock-free.

**Method 3: Rollback (Transaction-based)**

* **Step 1:** Identify the processes involved in the deadlock.
* **Step 2:** Rollback one or more processes to a previous safe state, effectively undoing their operations until they no longer hold resources.
* **Step 3:** Reassign resources as needed and allow processes to continue from their last safe state.



### **Procedure for Deadlock Detection and Resolution in a Multi-Process Operating System**

**1. Define Requirements**

1. **Identify the key components for deadlock management**:
   * **Processes** to be monitored.
   * **Resources** that are shared between processes.
   * **Conditions** that lead to deadlocks (e.g., circular wait).
2. **Determine the methods for deadlock detection**:
   * **Algorithms** to detect deadlocks (e.g., Wait-for Graph, Resource Allocation Graph).
   * **Thresholds** for resource allocation and process termination policies.

**2. Gather Components**

1. **Hardware/Software Requirements**:
   * **System Resources**: Memory, CPU time, I/O devices.
   * **Processes**: Various system or user processes.
   * **Operating System**: Should support multi-tasking and process management.
2. **Software Tools**:
   * **Programming Language** (e.g., C, C++, Java) to implement detection algorithms.
   * **Libraries** for system-level programming (e.g., POSIX threads for multi-process management).
   * **Algorithm Libraries** for cycle detection (e.g., Depth First Search).

**3. Design the System**

1. **Define Process and Resource Allocation**:
   * Create a model to represent processes and resources.
   * Use **Resource Allocation Graph (RAG)** or **Wait-for Graph (WFG)** to model resource requests.
2. **Design Data Structures**:
   * Use data structures like matrices or adjacency lists to track resource allocation and requests.
3. **Identify Deadlock Conditions**:
   * Define conditions under which a deadlock may occur, such as circular wait, hold and wait, no preemption, and mutual exclusion.

**4. Write the Code**

1. **Deadlock Detection**:
   * Implement the **Wait-for Graph** or **Resource Allocation Graph**.
   * Write algorithms to detect cycles in these graphs (e.g., using **Depth-First Search** or **Floyd-Warshall** for cycle detection).
2. **Deadlock Resolution**:
   * Implement methods for deadlock resolution like **process termination**, **resource preemption**, or **rollback**.
   * Define the conditions and strategies for each resolution method (e.g., choosing processes to terminate or selecting resources to preempt).
3. **Testing and Debugging**:
   * Use simulated data to test the system's ability to detect and resolve deadlocks.
   * Test different deadlock scenarios and adjust the algorithms if necessary.

**5. Upload and Test**

1. **Run the Program**:
   * Compile the code and run the system in a test environment.
2. **Simulate Processes and Resources**:
   * Simulate processes requesting and holding resources to create deadlock situations.
3. **Verify Deadlock Detection and Resolution**:
   * Ensure that the system correctly identifies deadlocks.
   * Check if the resolution techniques (e.g., process termination or resource preemption) resolve the deadlock and allow the system to continue operating.

**6. Optimize the System**

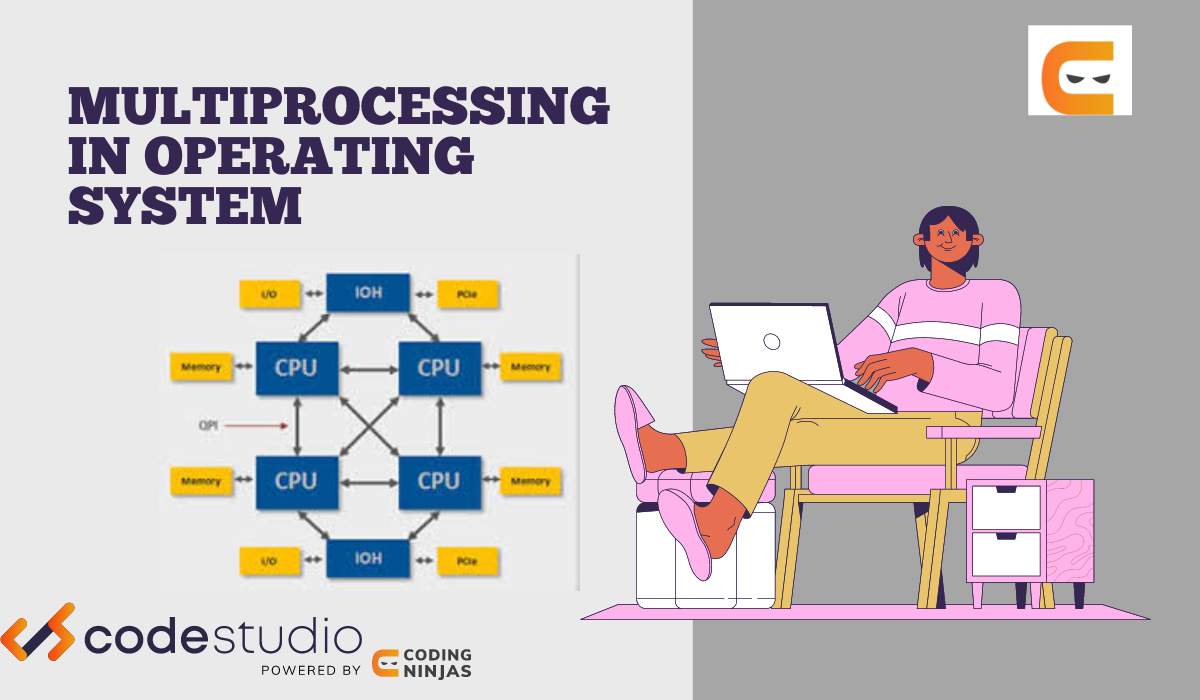
1. **Improve Efficiency**:
   * Fine-tune the algorithms for faster cycle detection and resolution.
2. **Scalability**:
   * Ensure that the system can handle large numbers of processes and resources efficiently.
3. **Enhance Recovery Strategies**:
   * Implement more advanced strategies for recovery, such as using a priority-based process selection for termination or more granular resource management.

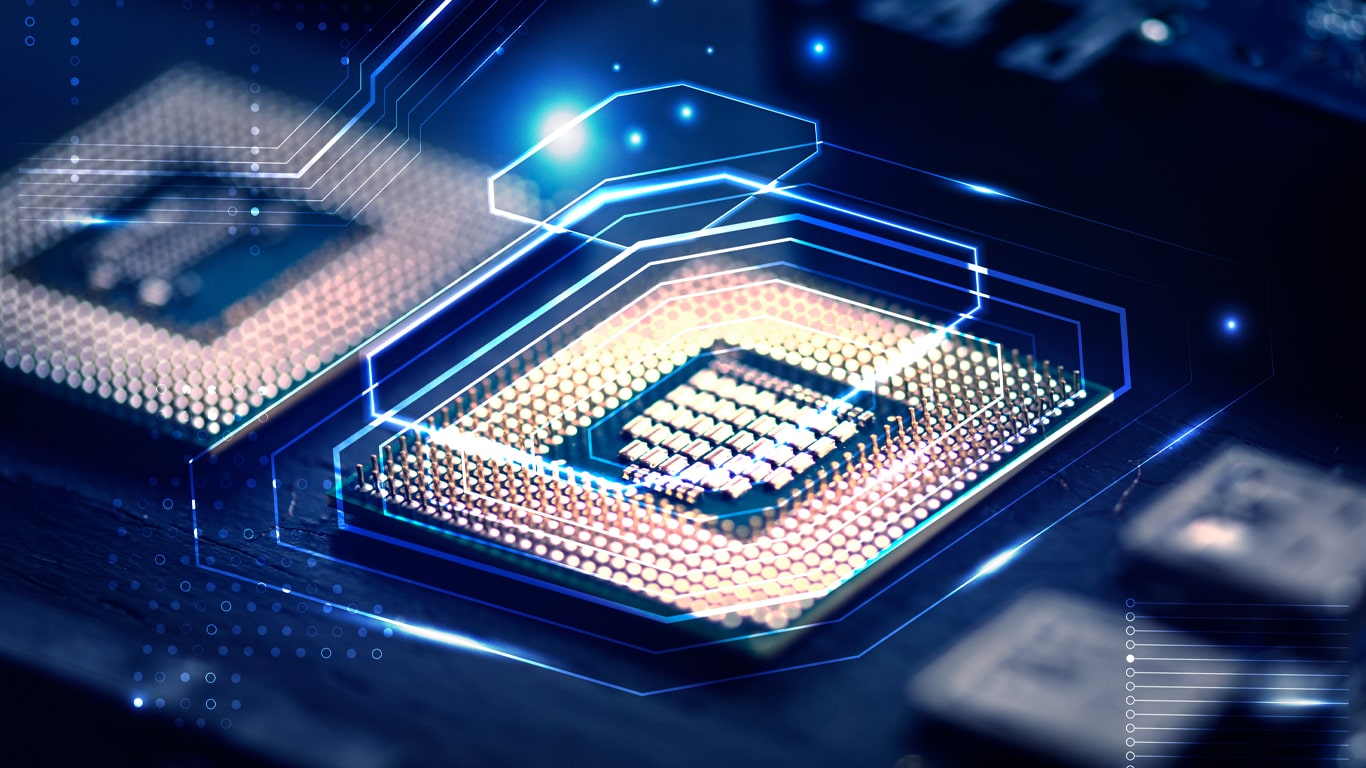
**7. Deploy the System**

1. **Integrate into Operating System**:
   * Integrate the deadlock detection and resolution system into the target operating system.
2. **Monitor in Real-Time**:
   * Continuously monitor processes and resources in a live environment to ensure deadlocks are detected early.
3. **Test in Real-World Scenarios**:
   * Deploy the system in various real-world scenarios to test its robustness and ensure it effectively resolves deadlocks.

**8. Maintenance**

1. **Regularly Update the System**:
   * Periodically check for bugs or inefficiencies in the detection and resolution mechanisms.
2. **Monitor System Performance**:
   * Monitor the operating system’s performance and adjust the algorithm as needed to ensure smooth operation.
3. **Add New Features**:
   * Implement additional features for deadlock prevention or enhanced detection (e.g., adaptive algorithms for changing system conditions).





**C program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

// Declare process and resource matrices

int allocation[MAX\_PROCESSES][MAX\_RESOURCES]; // Allocation matrix

int request[MAX\_PROCESSES][MAX\_RESOURCES]; // Request matrix

int available[MAX\_RESOURCES]; // Available resources

int process\_state[MAX\_PROCESSES]; // Process state (0 = not finished, 1 = finished)

// Function to detect deadlock using the Wait-for Graph and cycle detection

int is\_deadlock(int num\_processes, int num\_resources) {

int finish[MAX\_PROCESSES] = {0}; // 0 = not finished, 1 = finished

int safe\_count = 0;

// Attempt to finish processes if their requests can be satisfied

while (safe\_count < num\_processes) {

int found = 0;

for (int i = 0; i < num\_processes; i++) {

// Process is not yet finished and all its requests are less than or equal to available resources

if (finish[i] == 0) {

int can\_finish = 1;

for (int j = 0; j < num\_resources; j++) {

if (request[i][j] > available[j]) {

can\_finish = 0;

break;

}

}

// If the process can finish, simulate it by releasing resources

if (can\_finish) {

finish[i] = 1;

safe\_count++;

found = 1;

for (int j = 0; j < num\_resources; j++) {

available[j] += allocation[i][j]; // Release the resources

}

break;

}

}

}

// If no process can finish, then there is a deadlock

if (!found) {

return 1; // Deadlock detected

}

}

return 0; // No deadlock

}

// Function to simulate the allocation of resources to processes

void allocate\_resources(int num\_processes, int num\_resources) {

printf("Enter the Allocation Matrix:\n");

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &allocation[i][j]);

}

}

printf("Enter the Request Matrix:\n");

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &request[i][j]);

}

}

printf("Enter the Available Resources:\n");

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &available[j]);

}

}

// Function to resolve the deadlock by terminating one process

void resolve\_deadlock(int num\_processes) {

// In this simple implementation, we randomly terminate one process

int process\_to\_terminate = rand() % num\_processes;

printf("Deadlock detected! Terminating process %d to resolve the deadlock.\n", process\_to\_terminate);

// Terminate the process (mark it as finished)

for (int j = 0; j < MAX\_RESOURCES; j++) {

available[j] += allocation[process\_to\_terminate][j]; // Release its allocated resources

allocation[process\_to\_terminate][j] = 0; // Remove allocations

request[process\_to\_terminate][j] = 0; // Clear requests

}

}

// Function to print the matrices

void print\_matrices(int num\_processes, int num\_resources) {

printf("Allocation Matrix:\n");

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

printf("%d ", allocation[i][j]);

}

printf("\n");

}

printf("Request Matrix:\n");

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

printf("%d ", request[i][j]);

}

printf("\n");

}

printf("Available Resources:\n");

for (int j = 0; j < num\_resources; j++) {

printf("%d ", available[j]);

}

printf("\n");

}

int main() {

int num\_processes, num\_resources;

printf("Enter number of processes: ");

scanf("%d", &num\_processes);

printf("Enter number of resources: ");

scanf("%d", &num\_resources);

if (num\_processes > MAX\_PROCESSES || num\_resources > MAX\_RESOURCES) {

printf("Exceeded maximum number of processes or resources.\n");

return -1;

}

// Initialize the process state (0 means the process is not finished)

for (int i = 0; i < num\_processes; i++) {

process\_state[i] = 0; // Not finished

}

// Allocate resources

allocate\_resources(num\_processes, num\_resources);

// Print matrices

print\_matrices(num\_processes, num\_resources);

// Check for deadlock

if (is\_deadlock(num\_processes, num\_resources)) {

printf("Deadlock detected!\n");

resolve\_deadlock(num\_processes); // Resolve the deadlock

} else {

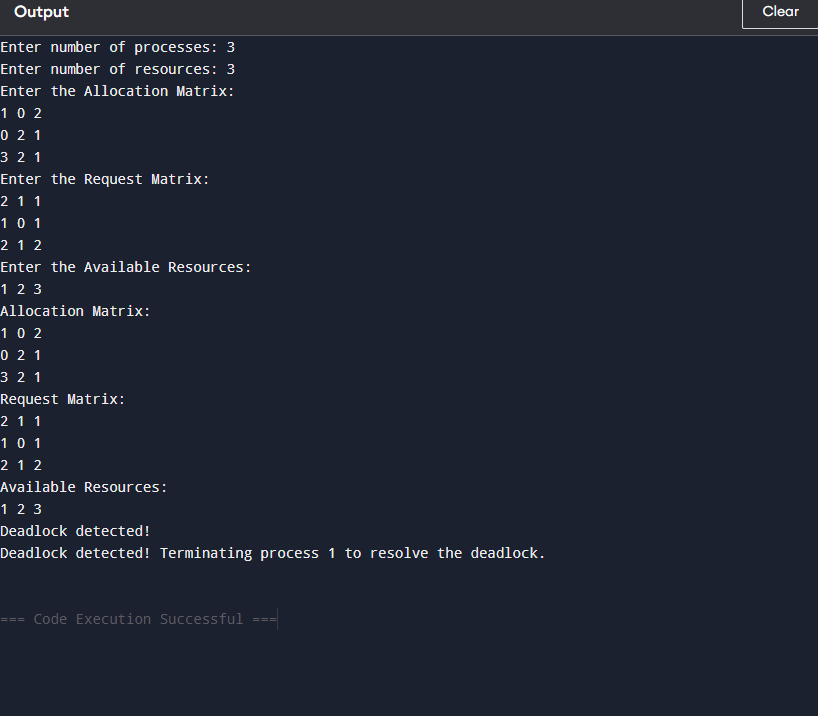
printf("No deadlock detected.\n");

}

return 0;

}

**Out put**

****

**Conclusion:**

In this project, we implemented a basic **Deadlock Detection and Resolution System** for a multi-process environment. The system uses an approach where it checks for deadlocks by simulating resource allocation and identifying situations where processes cannot proceed due to circular dependencies (waiting for resources held by other processes). If deadlock is detected, the system resolves it by terminating a randomly chosen process to release resources and allow the remaining processes to continue.

The key points of the implementation include:

1. **Deadlock Detection**: We used a **Wait-for Graph** model to simulate how processes wait for resources, and we detected deadlocks by checking for cycles in this graph. If no process can finish because it is waiting for resources held by other processes, a deadlock is detected.
2. **Deadlock Resolution**: In the event of a deadlock, the system terminates one process to free up resources and break the deadlock cycle. While this is a simplistic approach, it serves as a basic way to handle the deadlock situation in small systems.
3. **Resource Allocation Simulation**: The system allows for user input of the **allocation** and **request matrices**, enabling a clear view of how resources are distributed and requested among processes. This interactive nature helps in testing various deadlock scenarios.
4. **Scalability and Limitations**: This program works well for small-scale systems with a limited number of processes and resources. However, for more complex, real-world applications, a more sophisticated method like the **Banker's Algorithm** or **Priority-based Termination** would be necessary for more efficient deadlock handling.