Deadlock Detection and Resolution System: A Three-Tier Implementation Using Banker's Algorithm and Resource Allocation Graph Analysis

Raman Luhach Rachit Kumar Harshal Nerpagar

Department of Computer Science and Engineering

Academic Year 2025–2026

Abstract

Deadlock is a critical challenge in modern operating systems where concurrent processes compete for finite shared resources. When a set of processes enters a circular wait—each holding resources required by the next—the entire subset becomes permanently blocked, degrading system throughput and reliability. This paper presents a comprehensive *Deadlock Detection and Resolution System* implemented as a three-tier application. The core detection engine, written in C, implements Dijkstra's Banker's Algorithm to classify system states as safe or deadlocked and constructs a Resource Allocation Graph (RAG) with DFS-based cycle detection to independently confirm circular wait conditions. A RESTful API layer built with Express.js and TypeScript exposes six stateless endpoints for detection, step-by-step algorithm execution, deadlock resolution via victim process termination, and resource request simulation for deadlock avoidance. A React-based web frontend provides interactive matrix configuration, real-time RAG visualization using the React Flow library, and a step-by-step walkthrough mode that enables users to observe each iteration of the Banker's Algorithm. The system is validated against classical test vectors from operating systems literature and demonstrates correct identification of safe sequences, accurate deadlock detection, and successful resolution through minimum-cost process termination. The complete implementation spans 3,482 lines of source code across 26 files and supports configurations of up to 10 processes and 10 resource types.

Keywords—Deadlock Detection, Banker's Algorithm, Resource Allocation Graph, Operating Systems, Process Synchronization, Cycle Detection, Deadlock Resolution, Three-Tier Architecture

I. INTRODUCTION

In multiprogramming operating systems, multiple processes execute concurrently and compete for a finite pool of shared resources such as CPU cycles, memory pages, I/O devices, files, and synchronization primitives. While resource sharing enables higher system utilization and throughput, it introduces the possibility of *deadlock*—a state in which a set of processes becomes permanently blocked because each process holds at least one resource and waits for another resource held by a different process in the set [1].

Coffman et al. [2] established that deadlock arises if and only if four conditions hold simultaneously: (1) *mutual exclusion*, where at least one resource is non-shareable; (2) *hold and wait*, where a process holds resources while requesting additional ones; (3) *no preemption*, where resources cannot be forcibly reclaimed; and (4) *circular wait*, where a circular chain of processes exists, each waiting for a resource held by the next. Breaking any single condition prevents deadlock.

Operating systems employ four principal strategies to handle deadlock: prevention, avoidance, detection-and-recovery, and ignorance (the Ostrich Algorithm). Prevention imposes structural constraints to negate one or more Coffman conditions but restricts concurrency. Avoidance algorithms such as the Banker's Algorithm [3] dynamically verify that each resource grant maintains a *safe state* but incur overhead on every allocation request. Detection-and-recovery permits deadlocks to occur, periodically identifies them, and initiates corrective action—offering higher resource utilization at the cost of recovery complexity. Most production operating systems, including Linux and Windows, adopt the Ostrich approach for general resources due to the rarity and high overhead of prevention [1].

This paper presents a *Deadlock Detection and Resolution System* that implements the detection-and-recovery strategy through a modular three-tier architecture. The system's contributions include:

- A C-language implementation of the Banker's Algorithm for deadlock detection with safe sequence computation and a DFS-based Resource Allocation Graph (RAG) cycle detection module.
- A TypeScript REST API layer providing stateless endpoints for detection, step-by-step algorithm execution, resolution, and avoidance simulation.
- A React web frontend with interactive matrix editing, real-time graph visualization, and an educational step-by-step walkthrough mode.
- Validation against classical operating systems test vectors demonstrating correct behavior in both safe-state and deadlock scenarios.

II. RELATED WORK

A. Banker's Algorithm

Dijkstra introduced the Banker's Algorithm in 1965 as a deadlock avoidance technique [3]. The algorithm models the operating system as a banker managing limited capital (resources) across multiple customers (processes) with declared credit limits (maximum resource needs). Before each allocation, the banker verifies that granting the request preserves a safe state—one in which every process can complete in some sequential order. Habermann [4] later formalized the safety criterion and provided proofs of correctness. Our implementation uses the detection variant of the Banker's Algorithm, which operates on the current allocation state without requiring per-request safety checks.

B. Resource Allocation Graph

Holt [5] proposed the Resource Allocation Graph as a visual model for process-resource dependencies. In this directed graph, process and resource nodes are connected by *request edges* (process → resource) and *assignment edges* (resource → process). For single-instance resources, a cycle in the RAG is both necessary and sufficient for deadlock. For multi-instance resources, cycles are necessary but not sufficient, and the Banker's Algorithm must be used for definitive detection [1]. Our system implements both approaches: RAG-based cycle detection for visual confirmation and the Banker's Algorithm for authoritative classification.

C. Deadlock Recovery

Recovery from deadlock typically proceeds through process termination or resource preemption [1]. Process termination selects a *victim* process and aborts it, releasing its held resources to the available pool. Victim selection criteria include minimum cost, minimum resources held, and process priority. Resource preemption rolls back a process to a checkpoint and reallocates its resources. Our system implements the minimum-resource victim selection heuristic, which minimizes the disruption caused by termination.

III. SYSTEM DESIGN

A. Architecture Overview

The system follows a three-tier client-server architecture with clean separation of concerns between the presentation layer, business logic layer, and core algorithm engine. This design enables independent development, testing, and deployment of each tier while maintaining a unified data model across all layers.

Layer	Technology	Responsibility
Core Engine	C (C99, GCC)	Banker's Algorithm, RAG, Resolution
API Server	Express.js, TypeScript	RESTful endpoints, Validation
Frontend	React 19, Vite, TypeScript	Interactive UI, Visualization

TABLE I. Three-tier architecture components.

B. Data Model

The system's state is represented by the *SystemState* structure, which encapsulates the complete resource allocation context. The model supports up to 10 processes (P) and 10 resource types (R). Four matrices and one vector define the state:

• **Available[*j*]**: number of free instances of resource type *j*.

• **Allocation[*i*][*j*]**: instances of resource *j* currently held by process *i*.

• **MaxNeed[*i*][*j*]**: maximum demand of process *i* for resource *j*.

• **Need[*i*][*j*]**: remaining demand, computed as MaxNeed[*i*][*j*] − Allocation[*i*][*j*].

Detection produces a *DetectionResult* containing a boolean deadlock flag, the list of deadlocked process indices, and the safe sequence (when one exists). This result is consumed by the frontend for visualization and by the resolution module for victim selection.

C. API Design

The Express.js API server exposes six RESTful endpoints, all accepting and returning JSON. Every endpoint is stateless—the full system state is transmitted with each request, eliminating server-side session management. Table II enumerates the endpoints.

Method	Endpoint	Function
POST	/api/detect	Full Banker's Algorithm
POST	/api/detect/step	Single algorithm iteration
POST	/api/rag	Build RAG nodes and edges
POST	/api/resolve	Terminate victim, return new state
POST	/api/simulate-request	Avoidance simulation (dry run)
POST	/api/export	Validate and echo state as JSON

TABLE II. REST API endpoint specification.

IV. ALGORITHM DESIGN

A. Banker's Algorithm for Detection

The core detection algorithm is a direct implementation of the Banker's safety algorithm adapted for deadlock detection. Given the current system state, it determines whether a safe execution sequence exists for all processes. Algorithm 1 presents the pseudocode.

Algorithm 1: Banker's Deadlock Detection

```
Input: SystemState S
Output: DetectionResult R

Need[i][j] ← MaxNeed[i][j] - Allocation[i][j]
Work[j] ← Available[j]  ∀j
Finish[i] ← false  ∀i
repeat
  found ← false
  for each process i where Finish[i] = false do
    if Need[i][j] ≤ Work[j] ∀j then
      Work[j] ← Work[j] + Allocation[i][j]  ∀j
      Finish[i] ← true
      Append i to SafeSequence
      found ← true
 until found = false
if ∃i : Finish[i] = false then
  R.is_deadlocked ← true
  R.deadlocked ← {i | Finish[i] = false}
else
  R.safe_sequence ← SafeSequence
```

The algorithm operates in $O(P^2 \times R)$ time in the worst case, where P is the number of processes and R is the number of resource types. In each outer iteration, the algorithm scans all unfinished processes and selects one whose remaining need can be satisfied by the current work vector. Upon selection, the process's allocated resources are released back to the work pool, simulating process completion.

B. RAG Cycle Detection

The Resource Allocation Graph is constructed by creating two node types: process nodes (indices 0 to P–1) and resource nodes (indices P to P+R–1). Assignment edges are added from each resource node to each process node holding instances of that resource (Allocation[i][j] > 0). Request edges are added from each process node to each resource node for which the process has outstanding need (Need[i][j] > 0).

Cycle detection is performed using Depth-First Search (DFS) with a recursion stack. Algorithm 2 traverses all nodes; if a DFS path revisits a node currently on the recursion stack, a back edge (and hence a cycle) is detected. The time complexity is $O(V + E)$ where $V = P + R$ and E is the number of edges.

Algorithm 2: DFS Cycle Detection in RAG

```
Input: RAG G = (V, E)
Output: boolean (cycle exists)

1: visited[v] ← false, recStack[v] ← false  ∀v
2: for each node v ∈ V do
3:   if not visited[v] then
4:     if DFS_Cycle(v) then return true
5: return false

DFS_Cycle(v):
6:   visited[v] ← true; recStack[v] ← true
7:   for each neighbor u of v do
8:     if not visited[u] and DFS_Cycle(u) then
9:       return true
10:    if recStack[u] then return true
11:  recStack[v] ← false
12:  return false
```

C. Deadlock Resolution

When deadlock is detected, the resolution module selects a victim process for termination using a minimum-cost heuristic. The victim is the deadlocked process holding the fewest total allocated resource instances, computed as $\sum_j$ Allocation[*i*][*j*]. This heuristic minimizes the number of resources that must be reallocated and reduces the impact on overall system throughput.

Upon termination, all resources held by the victim are released back to the Available vector, and its Allocation and MaxNeed rows are zeroed. The detection algorithm is then re-executed on the modified state. If deadlock persists, additional victims may be selected iteratively until the system reaches a safe state.

D. Resource Request Simulation

The simulation module implements a deadlock avoidance check as a dry run. Given a request specifying a process index, resource index, and amount, the module creates a temporary copy of the system state, grants the request by incrementing the allocation and decrementing the available vector, and then runs the full Banker's detection algorithm on the modified state. If the resulting state is safe, the request would be granted; otherwise it would be blocked. This implements the classical Banker's avoidance strategy without modifying the actual system state.

V. IMPLEMENTATION

A. Core Engine (C)

The core detection engine is implemented in 965 lines of ANSI C (C99 standard) compiled with GCC. It is organized into three modules: *main.c* (329 lines) provides the interactive console driver with menu-based navigation and predefined sample scenarios; *deadlock_detector.c* (306 lines) implements the Banker's Algorithm, need matrix computation, and deadlock resolution; and *rag.c* (184 lines) constructs the Resource Allocation Graph and performs DFS-based cycle detection. Header files (*deadlock_detector.h* and *rag.h*) define the SystemState, DetectionResult, and RAG data structures.

The C implementation uses only standard library headers (stdio.h, stdlib.h, string.h, stdbool.h), ensuring portability across platforms. Constants MAX_PROCESSES and MAX_RESOURCES (both set to 10) bound the matrix dimensions. The build system uses GNU Make with compiler flags *-Wall -Wextra -std=c99* to enforce strict warning compliance.

B. API Server (TypeScript)

The API layer is implemented in 830 lines of TypeScript running on Node.js with Express.js v5. The Banker's Algorithm is ported line-for-line from the C implementation to ensure behavioral equivalence. The server exposes six endpoints (Table II), all operating statelessly over JSON payloads. Cross-Origin Resource Sharing (CORS) is configured to accept requests from the frontend development server at ports 5173 and 5174.

A notable feature is the step-by-step endpoint (*/api/detect/step*), which executes a single iteration of the Banker's Algorithm and returns the updated Work vector, Finish array, and a human-readable explanation string. The frontend calls this endpoint repeatedly to animate the algorithm. Input validation is performed at the API boundary using dedicated validation functions that verify matrix dimensions, non-negative values, and the constraint Allocation[*i*][*j*] ≤ MaxNeed[*i*][*j*] for all *i*, *j*.

C. Frontend (React)

The frontend is built with React 19 and Vite, comprising 1,687 lines across 18 TypeScript/TSX files. The application uses React Router for client-side navigation across five pages: a landing page with an OS glossary, a configuration and detection page, a standalone RAG visualization page, a step-by-step algorithm walkthrough page, and a resource request simulation page.

Global state management is handled through React Context (AppContext), which maintains the current system configuration, detection results, and the highlighted process index for cross-component synchronization. The *SystemConfigForm* component (343 lines) provides dynamic matrix editors that resize when the user changes the process or resource count, with real-time validation ensuring that allocation values do not exceed maximum needs.

The RAG visualization uses the *@xyflow/react* library (React Flow v12) to render an interactive directed graph. Process nodes are displayed as blue circles and resource nodes as green squares. Deadlocked processes are highlighted in red, and the currently active process during step-by-step execution is highlighted in orange with a glowing border. Request edges are drawn as animated dashed orange arrows, and assignment edges as solid green arrows. Users can drag nodes and pan the canvas.

D. Source Code Metrics

Layer	Files	Lines	Language
Core Engine	5	965	C (C99)
API Server	3	830	TypeScript
Frontend	18	1,687	TypeScript/TSX
Total	26	3,482	—

TABLE III. Source code metrics by layer.

Component	Version	Purpose
GCC / C99	System	Core algorithm compilation
Node.js	20+	JavaScript runtime for API
Express.js	5.2	HTTP server framework
TypeScript	5.9	Type-safe development
React	19.2	Frontend UI library
Vite	7.3	Frontend build tool
React Flow	12.10	Graph visualization
React Router	7.13	Client-side routing

TABLE IV. Technology stack with versions.

VI. RESULTS AND ANALYSIS

A. Test Case 1: Safe State

The first test vector represents the classical Banker's Algorithm example from Silberschatz et al. [1] with five processes and three resource types. The available vector is [3, 3, 2]. Table V shows the allocation and maximum need matrices.

Process	Alloc (R0,R1,R2)	Max (R0,R1,R2)	Need (R0,R1,R2)
P0	0, 1, 0	7, 5, 3	7, 4, 3
P1	2, 0, 0	3, 2, 2	1, 2, 2
P2	3, 0, 2	9, 0, 2	6, 0, 0
P3	2, 1, 1	2, 2, 2	0, 1, 1
P4	0, 0, 2	4, 3, 3	4, 3, 1

TABLE V. Safe state test configuration (Available = [3, 3, 2]).

The system correctly identifies this configuration as a **safe state** and computes the safe sequence P1 → P3 → P4 → P0 → P2. The step-by-step mode traces through five iterations: P1 is selected first because its Need [1, 2, 2] ≤ Work [3, 3, 2]; after P1 completes, Work becomes [5, 3, 2], enabling P3 (Need [0, 1, 1]), and so forth until all processes finish. The RAG module confirms the absence of cycles in the corresponding graph.

B. Test Case 2: Deadlock State

The second test vector creates a deadlock scenario with four processes, three resource types, and Available = [0, 0, 0]. Every process requires at least one additional unit of each resource type (Need = [1, 1, 1] for all), but zero resources are free.

Process	Alloc (R0,R1,R2)	Max (R0,R1,R2)	Need (R0,R1,R2)
P0	1, 0, 1	2, 1, 2	1, 1, 1
P1	1, 1, 0	2, 2, 1	1, 1, 1
P2	0, 1, 1	1, 2, 2	1, 1, 1
P3	1, 0, 0	2, 1, 1	1, 1, 1

TABLE VI. Deadlock state test configuration (Available = [0, 0, 0]).

The system correctly identifies all four processes as **deadlocked**. No process can be satisfied because Need[*i*] > Work = [0, 0, 0] for every *i*. The RAG module detects a cycle confirming the circular wait condition: P0 → R0 → P1 → R1 → P2 → R2 → P0.

C. Deadlock Resolution

When resolution is invoked on the deadlock state, the system selects P3 as the victim because it holds the fewest resources ($\sum$ Allocation[3] = 1 + 0 + 0 = 1, the minimum among deadlocked processes). After terminating P3, Available increases to [1, 0, 0]. Re-running detection on the modified state may require further victim terminations depending on the specific configuration, demonstrating the iterative nature of resolution.

D. Performance

Table VII summarizes the measured response times for each API endpoint under the maximum supported configuration (10 processes, 10 resources). All operations complete in under 20 milliseconds, confirming the feasibility of real-time interactive use.

Endpoint	Complexity	Response Time
/api/detect	$O(P^2R)$	< 10 ms
/api/detect/step	$O(PR)$	< 5 ms
/api/rag	$O(PR)$	< 5 ms
/api/resolve	$O(P^2R)$	< 20 ms
/api/simulate-request	$O(P^2R)$	< 10 ms

TABLE VII. API performance (P = 10, R = 10).

E. Request Simulation

The avoidance simulation was validated by attempting to grant P0 two units of R1 in the safe-state scenario. The system creates a temporary state with the modified allocation, runs the Banker's Algorithm, and reports that the grant maintains a safe state. Conversely, requesting an amount that would exceed a process's maximum need or the available resources is rejected with a descriptive error message prior to running the algorithm.

VII. EDUCATIONAL FEATURES

A key design goal of this system is its pedagogical value. Several features specifically support learning and demonstration of OS concepts:

- **Step-by-step mode:** Users observe each iteration of the Banker's Algorithm, seeing which process is selected, why

its Need can be satisfied, how the Work vector updates, and how the safe sequence is built incrementally.

- **Interactive RAG visualization:** The React Flow graph provides drag-and-drop nodes, animated edges, and color-coded highlighting that makes the abstract graph structure tangible.
- **Predefined scenarios:** Two built-in test cases (safe and deadlock) allow instant demonstration without manual matrix entry.
- **Export/import:** System states can be saved as JSON files and loaded later, supporting reproducible experiments and assignment-based learning.
- **OS glossary:** The landing page defines eight key terms (Allocation Matrix, Max Need, Need, Available Vector, Safe Sequence, RAG, Request Edge, Assignment Edge) for quick reference.

VIII. CONCLUSION

This paper presented a three-tier Deadlock Detection and Resolution System that bridges the gap between theoretical operating systems concepts and practical interactive visualization. The system implements the Banker's Algorithm for deadlock detection with safe sequence computation, DFS-based cycle detection on the Resource Allocation Graph for visual confirmation of circular wait conditions, and a minimum-cost victim selection strategy for deadlock resolution.

The implementation demonstrates that classical OS algorithms can be effectively delivered through modern web technologies. The three-tier architecture—with a C core engine, TypeScript REST API, and React frontend—provides clean separation of concerns while enabling rich interactive features such as step-by-step algorithm animation, real-time graph visualization, and deadlock avoidance simulation.

Validation against standard test vectors from operating systems literature confirms correct behavior in both safe and deadlocked configurations. The system supports configurations of up to 10 processes and 10 resource types with sub-20ms response times, making it suitable for real-time educational use.

IX. FUTURE WORK

Several directions exist for extending this work:

- **Distributed deadlock detection:** Extending the system to detect deadlocks in distributed environments using probe-based or edge-chasing algorithms.
- **Wait-for graph:** Implementing the Wait-for Graph as an alternative detection mechanism for single-instance resource systems.
- **Algorithm animation:** Adding frame-by-frame animation of the DFS traversal during RAG cycle detection to further enhance educational value.
- **Multi-instance RAG analysis:** Implementing the knot detection algorithm for accurate deadlock determination in multi-instance RAGs.
- **Real-time OS integration:** Interfacing with a real operating system's process table and resource descriptors to detect actual system deadlocks.

. REFERENCES

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[2] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, Jun. 1971.

[3] E. W. Dijkstra, "Cooperating sequential processes," Technical Report EWD-123, Eindhoven University of Technology, 1965.

[4] A. N. Habermann, "Prevention of system deadlocks," *Communications of the ACM*, vol. 12, no. 7, pp. 373–377, Jul. 1969.

[5] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179–196, Sep. 1972.

[6] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Pearson, 2014.

[7] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Hoboken, NJ, USA: Pearson, 2018.

[8] M. Singhal, "Deadlock detection in distributed systems," *Computer*, vol. 22, no. 11, pp. 37–48, Nov. 1989.

[9] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.