

Project Report: SafeVax – Deadlock-Free Vaccine Distribution System

1. Description of the Project

Objective:

The primary objective of this project is to design and simulate a robust, concurrent resource allocation system modeled after a vaccine distribution network during a pandemic. The system manages limited critical resources (Vaccines, Syringes, and Delivery Trucks) across multiple competing entities (Hospitals).

Core Concept:

The project implements an Operating System simulation where "Hospitals" act as processes and medical supplies act as system resources. The core goal is to prevent Deadlock—a situation where hospitals hold partial resources while waiting for others, causing the entire distribution chain to freeze. To achieve this, the system implements Dijkstra's Banker's Algorithm to ensure the system remains in a "Safe State" at all times.

2. Significance of the Project

Meaningfulness:

In real-world logistics, specifically healthcare supply chains, resources are finite and demand is unpredictable. If a distribution center allocates resources greedily (first-come, first-served) without foresight, it risks a "gridlock" where no hospital has enough supplies to treat patients, yet all supplies are distributed. This project demonstrates how computer science principles (process synchronization and deadlock avoidance) can theoretically optimize life-saving logistics.

Novelty:

While the Banker's Algorithm is a standard OS concept, this project introduces a unique "Tamper/Chaos" layer. Unlike static simulations, this project allows the user to dynamically trigger "Supply Chain Collapses" (halving inventory) or "Demand Surges" (spiking hospital needs) in real-time. This stress-tests the algorithm's adaptability, simulating the volatility of a real crisis environment.

3. Code Structure

The system is built using Python's threading library for concurrency and tkinter for the graphical interface. It follows a Model-View-Controller (MVC) hybrid pattern.

Systematic Structure Diagram

graph TD

User[User / GUI] -->|Toggle Safety / Trigger Crash| RM[Resource Manager (Monitor)]

H1[Hospital Thread 1] <-->|Request/Release| RM

H2[Hospital Thread 2] <-->|Request/Release| RM

H3[Hospital Thread 3] <-->|Request/Release| RM

RM -->|Push State Updates| Q[GUI Queue]

Q -->|Process Events| GUI[SafeVax GUI Display]

Component Explanation

1. ResourceManager (The Monitor/Banker):

- Acts as the central authority. It holds the total, available, allocated, and need matrices.
- Uses `threading.Lock()` to ensure thread safety during resource modification.

2. Hospital (The Process):

- Inherits from `threading.Thread`.
- Simulates a lifecycle: Idle \rightarrow Calculate Need \rightarrow Request Resources \rightarrow Treat Patients (Hold Resources) \rightarrow Release Resources.

3. SafeVaxGUI (The View):

- Visualizes the state of the system using a Grid layout.
- Polls a thread-safe `gui_queue` to update the UI without freezing the main application loop.
- Provides user controls for the "Danger Zone" (Tamper Controls).

4. Description of Algorithms

The project relies on **Dijkstra's Banker's Algorithm** for deadlock avoidance.

Logic:

When a hospital requests resources, the system does not grant them immediately, even if the resources are currently available in the warehouse. Instead, the system performs a simulation (a "provisional allocation"):

1. Assume the request is granted.
2. Update the Available, Allocation, and Need matrices.
3. Check if the system is in a **Safe State**.

Safe State Definition:

A state is safe if there exists at least one sequence of execution such that every hospital can eventually receive its maximum demand, finish its task, and return its resources to the pool.

5. Verification of Algorithms

Let us verify the algorithm with a simplified numerical example based on the logic in the code.

Scenario:

- **Total Resources:** 10 Vaccines.
- **Current Available:** 3 Vaccines.

Status of Hospitals:

- **H1:** Has 2, Needs 4 more (Max 6).
- **H2:** Has 4, Needs 2 more (Max 6).
- **H3:** Has 1, Needs 5 more (Max 6).

Request: H1 requests 1 Vaccine.

1. **Check 1 (Availability):** Request (1) \leq Available (3). *True*.
2. **Provisional State:**
 - Available becomes 2.
 - H1 Has 3, Needs 3.
3. **Safety Check (Can we finish everyone?):**
 - *Current Work (Available) = 2.*
 - *Can H1 finish? Need 3 > Work 2. (No).*
 - *Can H3 finish? Need 5 > Work 2. (No).*
 - *Can H2 finish? Need 2 \leq Work 2. (Yes).*
 - **Simulate H2 Finish:** H2 returns its holdings (4). New Work = 2 + 4 = 6.
 - **Simulate H1 Finish:** H1 returns holdings (3). New Work = 6 + 3 = 9.
 - **Result:** All True.

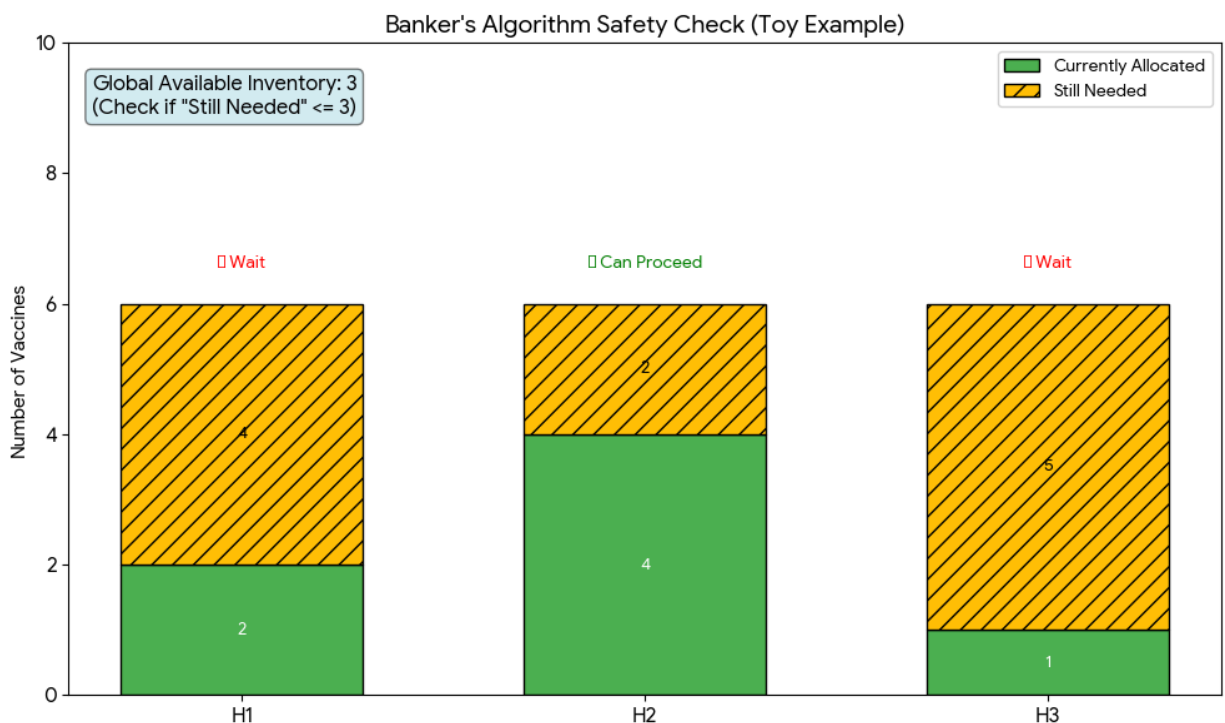
6. Functionalities

1. **Resource Management Dashboard:** Displays real-time counts of Vaccines, Syringes, and Trucks in the Global Warehouse.

2. **Concurrent Request Handling:** Supports multiple hospitals generating randomized demands simultaneously without data corruption (race conditions) due to threading locks.
3. **Safety Protocol Toggle:** Users can enable/disable the Banker's Algorithm dynamically.
 - *Enabled:* Requests are denied if they lead to an unsafe state.
 - *Disabled:* Requests are granted immediately if stock exists (risking deadlock).
4. **Chaos Engineering (Tamper Controls):**
 - **Supply Crash:** Instantly reduces global inventory by 50% to test scarcity handling.
 - **Demand Surge:** Randomly increases the "Max Need" of all hospitals, making the system harder to satisfy.

7. Execution Results and Analysis

Visualization:



This chart illustrates the decision-making process described in the **Verification of Algorithms** section. It shows why the algorithm allows Hospital 2 (H2) to proceed while others must wait.

- **Green Bar (Allocated):** Resources the hospital already holds.
- **Yellow Bar (Needed):** Additional resources required to finish the task.
- **Logic:** The system compares the **Yellow Bar (Need)** against the **Global Available Inventory (3)**.
 - **H1 (Need 4):** $4 > 3 \rightarrow$ **WAIT** (Cannot be satisfied).

- **H3 (Need 5):** $5 > 3 \rightarrow$ **WAIT** (Cannot be satisfied).
- **H2 (Need 2):** $2 \leq 3 \rightarrow$ **PROCEED** (Safe to grant).

Once H2 finishes, it returns its 4 held items, increasing the Global Available to $3 + 4 = 7$. Then, H1 (Need 4) and H3 (Need 5) can be satisfied using the new inventory.

The GUI provides a color-coded status update.

- **Green Status:** "Allocated (Safe)" or "Treating Patients."
- **Red Status:** "Denied (Unsafe State)."
- **Logs:** A scrolling log window details specific decisions (e.g., "🔴 [UNSAFE] Hospital-1 DENIED by Banker's Algo").

Analysis of Results:

1. **With Safety Enabled:** The system runs continuously. You will observe frequent "Waiting" or "Denied" statuses. This appears inefficient locally (hospitals wait longer), but globally, the system never freezes. The "Available" resources never drop to zero in a way that prevents the "closest to finish" hospital from completing.
2. **With Safety Disabled:** Initially, the system runs faster. However, as "Allocated" numbers rise, the "Available" resources drop near zero. Eventually, all 5 hospitals may display "Waiting for Stock," and none will ever release their resources because they are all waiting for items held by others. This is a complete **System Freeze (Deadlock)**.
3. **Impact of Supply Crash:** When the button is pressed, the log shows "Inventory halved." The Banker's Algorithm immediately becomes more conservative, denying more requests to preserve the reduced buffer needed to ensure safety.

8. Conclusions

Summary of Findings:

The project successfully demonstrates that while deadlock avoidance algorithms (Banker's) introduce overhead and request denials, they are essential for system stability in resource-constrained environments. The "Safety Off" mode proved that greedy allocation inevitably leads to system failure.

Project Issues:

- **Starvation:** In a very tight supply scenario, a hospital with high demands might be denied repeatedly while smaller hospitals are serviced, leading to starvation.

- **Conservative Nature:** The Banker's Algorithm is pessimistic; it may deny a request that *could* have been safe because it assumes the worst-case scenario (all hospitals requesting max simultaneously).

This project applied critical OS concepts: Concurrency (Threads), Synchronization (Mutex/Locks), Deadlock Characterization (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait), and Deadlock Avoidance (Banker's Algorithm)