Group Project 3: Banker’s Algorithm

Course: CECS 326 SEC 02 - Operating Systems
Project: Banker’s Algorithm
Team Members:

- Teammate 1: Phu Quach (ID: 029475548)
- Teammate 2: Ryan Tran (ID: 031190716)

Date: November 7, 2025

I . Summary

We built a Python program to demonstrate the Banker's Algorithm for deadlock avoidance in resource allocation systems. Our code demonstrates the core concepts of safe state detection, resource request validation, and dynamic resource allocation in a multi-process environment with multiple resource types. The system manages 5 processes competing for 3 types of resources, ensuring deadlock-free execution through state validation.

YouTube Demo: https://www.youtube.com/watch?v=Q5750ugkNyg
GitHub: https://github.com/harvest7777/proj3

II. Project Implementation

We simulate the Banker’s Algorithm using 5 processes from P0 to P4 competing for 3 types of resources R0, R1 and R2. Available vector keeps track of available instances of each resource type and is initialized to [3, 3, 2] at program start. In addition, Maximum matrix defines a 5x3 matrix where each row represents a process and each column represents a resource type. It let us know that the maximum resources each process might need during entire execution. For instance, process P0 has a maximum need [7, 5, 3] which meaning it needs 7 units of R0, 5 units of R1, and 3 units of R2. Allocation matrix tracks the resources that are assigned to each process and Need matrix is calculated as the difference between Maximum and Allocation by Need = Max - Allocation, it tells how many more resources each process needs to complete execution. As a result, a finished Boolean array tracks which processes have completed the execution.

Moreover, the **can_finish(process_index, available, need)** function checks if a specific process can complete with the currently available resources. It loops through resource type, then they checks if the process's need for that resource is less than or equal to what's available. If all the resource requirements is good then it returns true otherwise returns false. It prevents us from partially allocating resources leads to deadlock. The **find_process_to_finish(available, need, finished)** function implements the process selection scanning through all processes to find one that hasn't finished yet and can complete with the current available resources. It checks each unfinished process using **can_finish()**, returning the index of the first eligible process or -1 if no such process exists, which would specify an unsafe state.

The **finish_process()** function handles the completion of a process by simulating resource reclamation. When the safety simulation (or the special case in the request protocol) treats a process as completed, this function models releasing the process's current allocation back to the system. It adds Allocation[process_index] to the temporary Available, sets Need[process_index] to 0, and sets Finished[process_index] = True so it cannot be selected again. This implementation does not zero the Allocation row itself; Finished=True with Need=0 indicates completion.

The **request_resources(process_index, request, available, need, allocation, finished)** function implements the resource request protocol with safety validation. The routine first enforces two necessary conditions: (1) request ≤ Need[process_index], and (2) request ≤ Available. If either check fails, it raises an error without changing state. If request == Need[process_index], this implementation treats the process as immediately finishing and calls finish_process. Important: in this equal-Need branch, the code does not subtract the request from Available nor add it back later; only the process's current Allocation is released. Otherwise (when checks pass and request < Need), the allocation committed immediately: Available decreases by request, Allocation[process_index] increases by request, and Need[process_index] decreases by request. After committing (or after the equal-Need finish), the caller prints the safe sequence using **find_safe_sequence**. This version does not automatically roll back a committed allocation if the subsequent safety check yields no safe sequence

The **find_safe_sequence(available, need, allocation, finished)** function is responsible for the Banker's Algorithm's safety verification. The routine first creates deep copies of Available, Need, Allocation, and Finished so the real system state remains unchanged. It repeatedly calls **find_process_to_finish** to select a process that can complete, appends it to the sequence, and calls **finish_process** on the temporary copies to release its allocation. If a full sequence of length n is built, that sequence is returned; if at any point no unfinished process satisfies Need ≤ Available, function returns None.

In the main, we can choose to find a safe sequence, request resources for a specific process or exit the program. After a request is processed, the program prints "System in safe state" followed by the safe sequence; if find_safe_sequence returns None, the message still prints but the sequence shown is None. Users enter a process index and a space-separated request vector of length m.

III. Contribution

1. **Ryan:**

- Coding program, implemented the core functions and data structures
    - Implemented find_safe_sequence() function
    - Developed can_finish() and finish_process()
- Writing comment, and write a readme file (compilation and usage instructions)
- Record video

2. **Phu Quach :**

- Implemented resource request handling (request_resources)
- Tested the program thoroughly to find bugs and edge cases
- Written comprehensive documentation

IV. Conclusion

Our implementation shows the Banker's Algorithm's effectiveness in preventing deadlocks through resource management. The system identifies safe states, handles resource requests with validation, and maintains the system throughout operation.

IV. References

1. https://www.geeksforgeeks.org/operating-systems/bankers-algorithm-in-operating-system-2/
2. https://www.geeksforgeeks.org/dsa/bankers-algorithm-in-operating-system/
3. https://www.youtube.com/watch?v=7gMLNiEz3nw