UTM
UNIVERSITI TEKNOLOGI MALAYSIA
School of Professional and Continuing Education (SPACE)

Group 6

Banker's Algorithm for Deadlock Avoidance

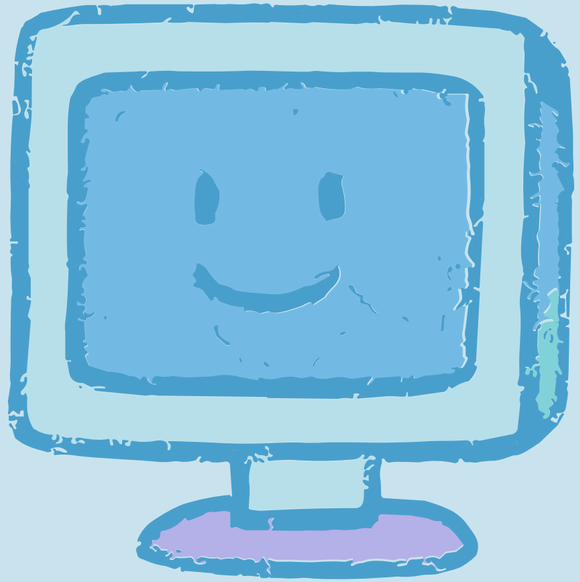
Lecturer: Dr Ismail

DSPD 2663: Operating System

Members: XuanHui,Engku,Azhani,Collin

Introduction

- Banker's Algorithm manages resources safely among processes.
- Works like a bank giving out loans only when it can still meet all customers' needs.
- Prevents deadlock by checking available, allocated, and needed resources before approval.



Problem Statement

- Many processes run together and need CPU time, memory, and files.
- Without careful management, the system may fall into deadlock — all processes wait forever.
- The challenge: how to allocate resources fairly while staying in a safe state.

Project Objectives

1. Understand how Banker's Algorithm avoids deadlock.
2. Learn safe resource allocation for multiple processes.
3. Design and implement a safety check program.
4. Test the algorithm with sample data.
5. Analyze performance and suggest improvements.

Design Overview

Core Data Structures:

Allocation[n][m]	current allocation
MaxNeed[n][m]	maximum resources needed
Available[m]	available resources
RemainNeed[n][m]	remaining need
Status[n]	process completion status

Algorithm Steps

Step 1: Initialize data (allocation, max need, available).

Step 2: Apply Safety Algorithm to check resource requests.

Step 3: Verify if all processes can complete safely.

Output:

✅ Safe Sequence → System is in safe state

❌ Unsafe → Possible deadlock

Code Overview

- Check whether each process's remaining need can be met by available resources.
- If safe, the process runs and releases its allocated resources.
- Form the safe sequence showing the correct execution order.

```
for (int i = 0; i < n; i++) {
    if (status[i]) continue;     // Skip completed process
    bool canAllocate = true;

    for (int j = 0; j < m; j++) {
        if (RemainNeed[i][j] > work[j]) {
            canAllocate = false; // Not safe to run yet
            break;
        }
    }
    if (canAllocate) {
        for (int j = 0; j < m; j++)
            work[j] += Allocate[i][j];  // Release resources
        status[i] = true;
        sequence += "P" + to_string(i) + " -> ";
    }
}
```

Test Case Example

- Given: n = 5, m = 3Input tables for Allocation and MaxNeed.
- Program Output:System is in safe state → Safe sequence displayed.

```
------------
this is safe

------------
7 2 5
------------
Processes | Allocation | MaxNeed      |Available     | RemainNeed   | status
P0        | 0 1 0      | 7 5 3        | 7 4 5        | 7 4 3        | true
P1        | 2 0 0      | 3 2 2        | 3 3 2        | 1 2 2        | true
P2        | 3 0 2      | 9 0 2        | 7 5 5        | 6 0 0        | true
P3        | 2 1 1      | 2 2 2        | 5 3 2        | 0 1 1        | true
P4        | 0 0 2      | 4 3 3        | 7 4 3        | 4 3 1        | true

------------
P1-> P3-> P4-> P0-> P2->
```

Performance Analysis

Case	Description	Time Complexity
Best	All processes safe in 1 pass	$O(n \times m)$
Worst	Only 1 process per pass	$O(n^2 \times m)$
Average	2–3 passes	$O(n \times m) \rightarrow O(n^2 \times m)$

Efficient for small systems but may slow down for large dynamic systems.

Limitations

- Static input only (no dynamic requests).
- Fixed number of processes and resources.
- No real-time monitoring or recovery.
- Text-based, no visual interface.

Suggested Improvements

1. Add dynamic request handling in runtime.
2. Use priority-based selection for better scheduling.
3. Parallel safety checking using OpenMP.
4. Add graphical visualization (Gantt chart / dashboard).

Conclusion

Banker's Algorithm ensures safe resource allocation by checking every request before approval.It effectively prevents deadlocks and helps maintain system stability.While limited for modern systems, it's essential for understanding OS resource management.

Code Demo