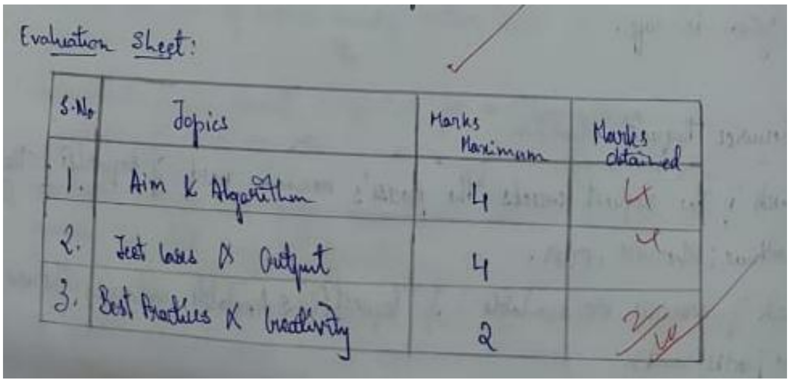
LMS Assignment Report

Department: Department of Computer Science & Engineering
Course: ICS1313 - Operating System Practices Laboratory
Assignment No: 8
Name: Simiyon Vinscent Samuel L
Reg No: 3122247001062

Assignment Title

Implementation of Banker's Algorithm for Deadlock Avoidance (Exercise 8)

Evaluation Sheet



Evaluation Sheet:

S.No	Topics	Marks Maximum	Marks Obtained
1.	Aim & Algorithm	4	4
2.	Test cases & Output	4	4
3.	Best Practices & Creativity	2	2

Aim / Learning Objectives

To develop a C program implementing the Banker's algorithm for deadlock avoidance, using a menu-driven interface to read data, print matrices, and compute a safe sequence if possible. Objectives include:

- Understanding deadlock avoidance through resource allocation checks.

- Implementing matrices for allocation, max, need, and available resources.

- Testing for safe and unsafe states using the safety algorithm.

- Applying OS concepts to prevent deadlocks in multi-process systems.

Algorithm

9/9/25

Experiment - 8

Aim: To develop a C program to implement the Banker's algorithm for deadlock avoidance.

Algorithm:

The Banker's Algorithm is made up of two parts:

1. Safety Algorithm
2. Resource Request Algorithm

→ Safety Algorithm

1. Initialize:
   - Work = Available
   - Finish [i] = false for all processes.
2. Look for a process Pi such that:
   - Finish [i] = false
   - Need[i] ≤ Work
3. If such a process is found:
   - Pretend to allocate resources to it: Work += Allocation [i]
   - Mark the process finished: Finish [i] = true
   - Repeat step 2 for remaining processes.
4. If all processes are marked finished (Finish[i] = true for all), the system is safe.

→ Resource Request Algorithm

1. Check if the request exceeds the process's maximum need: If Request[i] ≤ Need[i], continue; otherwise, error.
2. Check if resources are available: If Request[i] ≤ Available, continue; otherwise, the process waits.
3. Temporarily allocate the resources:
   - Available = Available − Request [i]
   - Allocation [i] = Allocation [i] + Request [i]
   - Need[i] = Need[i] − Request [i]
4. Run the Safety Algorithm:
   - If the new state is safe, grant the request.
   - If unsafe, roll back and make the process wait.

Test Cases

1. **Test Case 1: Safe State (Sample)**

•Input: Number of processes: 5; Number of resources: 3; Available: 3 3 2; Max: P0(7 5 3), P1(3 2 2), P2(9 0 2), P3(2 2 2), P4(4 3 3); Allocation: P0(0 1 0), P1(2 0 0), P2(3 0 2), P3(2 1 1), P4(0 0 2)

•Expected Output: Print Data shows matrices; Safety Sequence: e.g., P1 P3 P4 P0 P2 (or valid sequence)

2. **Test Case 2: Unsafe State**

•Input: Number of processes: 5; Number of resources: 3; Available: 2 1 0; Max: P0(7 5 3), P1(3 2 2), P2(9 0 2), P3(2 2 2), P4(4 3 3); Allocation: P0(0 1 0), P1(2 0 0), P2(3 0 2), P3(2 1 1), P4(0 0 2)

•Expected Output: Print Data shows matrices; Safety Sequence: No safe sequence exists (system is in unsafe state)

3. **Test Case 3: Minimal Resources Safe State**

•Input: Number of processes: 3; Number of resources: 2; Available: 1 1; Max: P0(2 2), P1(1 3), P2(3 1); Allocation: P0(1 0), P1(0 1), P2(0 0)

•Expected Output: Print Data shows matrices; Safety Sequence: e.g., P0 P1 P2 (or valid order)

C Code

```c
// C program for Banker's Algorithm with Menu-driven Interface
#include <stdio.h>

#define MAX_P 10
#define MAX_R 10

int main() {
    int n = 0, r = 0;
    int avail[MAX_R], alloc[MAX_P][MAX_R], max[MAX_P][MAX_R], need[MAX_P][MAX_R];
    char res_names[MAX_R] = {'A','B','C','D','E','F','G','H','I','J'};
    int data_read = 0; // Flag to check if data has been read

    while (1) {
        printf("\nBanker's Algorithm\n");
        printf("1. Read Data\n");
        printf("2. Print Data\n");
        printf("3. Safety Sequence\n");
        printf("4. Exit\n");
        printf("Enter the option: ");
        int option;
        scanf("%d", &option);

        if (option == 1) {
            // Read Data
            printf("Number of processes: ");
            scanf("%d", &n);
            printf("P0");
            for (int i = 1; i < n; i++) printf(", P%d", i);
            printf("\nNumber of resources: ");
            scanf("%d", &r);
```

```c
        for (int i = 0; i < r; i++) {
            printf("Number of Available instances of %c: ", res_names[i]);
            scanf("%d", &avail[i]);
        }
        for (int i = 0; i < n; i++) {
            printf("Maximum requirement for P%d: ", i);
            for (int j = 0; j < r; j++) {
                scanf("%d", &max[i][j]);
            }
        }
        for (int i = 0; i < n; i++) {
            printf("Allocated instances to P%d: ", i);
            for (int j = 0; j < r; j++) {
                scanf("%d", &alloc[i][j]);
            }
        }
        // Calculate Need matrix
        for (int i = 0; i < n; i++)
            for (int j = 0; j < r; j++)
                need[i][j] = max[i][j] - alloc[i][j];
        data_read = 1;
    } else if (option == 2) {
        if (!data_read) {
            printf("Please read data first (Option 1).\n");
            continue;
        }
        // Print Data
        printf("Pid\tAlloc\tMax\tNeed\tAvail\n");
        printf("\t");
        for (int j = 0; j < r; j++) printf("%c ", res_names[j]);
        printf("\t");
        for (int j = 0; j < r; j++) printf("%c ", res_names[j]);
        printf("\t");
        for (int j = 0; j < r; j++) printf("%c ", res_names[j]);
        printf("\t");
        for (int j = 0; j < r; j++) printf("%c ", res_names[j]);
        printf("\n");

        for (int i = 0; i < n; i++) {
            printf("P%d\t", i);
            for (int j = 0; j < r; j++) printf("%d ", alloc[i][j]);
            printf("\t");
            for (int j = 0; j < r; j++) printf("%d ", max[i][j]);
            printf("\t");
            for (int j = 0; j < r; j++) printf("%d ", need[i][j]);
            printf("\t");
            if (i == 0) // Show Avail only for first row
                for (int j = 0; j < r; j++) printf("%d ", avail[j]);
            printf("\n");
        }
    } else if (option == 3) {
        if (!data_read) {
            printf("Please read data first (Option 1).\n");
            continue;
```

```c
            }
            // Safety Sequence
            int work[MAX_R], finish[MAX_P], safe_seq[MAX_P], idx = 0;
            for (int i = 0; i < r; i++) work[i] = avail[i];
            for (int i = 0; i < n; i++) finish[i] = 0;

            // Find a safe sequence
            for (int count = 0; count < n; count++) {
                int found = 0;
                for (int p = 0; p < n; p++) {
                    if (!finish[p]) {
                        int can_allocate = 1;
                        for (int j = 0; j < r; j++) {
                            if (need[p][j] > work[j]) {
                                can_allocate = 0;
                                break;
                            }
                        }
                        if (can_allocate) {
                            for (int j = 0; j < r; j++)
                                work[j] += alloc[p][j];
                            safe_seq[idx++] = p;
                            finish[p] = 1;
                            found = 1;
                        }
                    }
                }
                if (!found) break;
            }

            printf("Display the Safety Sequence: ");
            if (idx == n) {
                for (int i = 0; i < n; i++)
                    printf("P%d ", safe_seq[i]);
                printf("\n");
            } else {
                printf("No safe sequence exists (system is in unsafe state).\n");
            }
        } else if (option == 4) {
            // Exit
            printf("Exiting...\n");
            break;
        } else {
            printf("Invalid option! Try again.\n");
        }
    }
    return 0;
}
```

Code Explanation

The code implements Banker's algorithm with a menu interface. Here's a detailed breakdown:

-**Headers and Defines**: Includes ` <stdio.h> ` for I/O. Defines MAX_P=10, MAX_R=10 for max processes/resources.

- **Main Function and Menu**: Loops with options: 1 (Read Data: input n, r, available, max, allocation; compute need), 2 (Print Data: display matrices in tabular format with resource names A-J), 3 (Safety Sequence: use work array, finish flags; find processes where need <= work, add allocation to work, collect sequence; check if all finished), 4 (Exit).

- **Safety Algorithm Logic**: Initializes work=available, finish=0. Loops to find allocatable processes, updates work, marks finish=1. If all finished, prints sequence; else, unsafe.

This prevents deadlocks by simulating allocations; without it, circular waits could occur.

Output Screenshots

Safe State –

```
ks_vijay-1401@DESKTOP-J8G3TP8:~$ cc bankers.c
ks_vijay-1401@DESKTOP-J8G3TP8:~$ ./a.out

Banker's Algorithm
1. Read Data
2. Print Data
3. Safety Sequence
4. Exit
Enter the option: 1
Number of processes: 5
P0, P1, P2, P3, P4
Number of resources: 3
Number of Available instances of A: 3
Number of Available instances of B: 3
Number of Available instances of C: 2
Maximum requirement for P0: 7 5 3
Maximum requirement for P1: 3 2 2
Maximum requirement for P2: 9 0 2
Maximum requirement for P3: 2 2 2
Maximum requirement for P4: 4 3 3
Allocated instances to P0: 0 1 0
Allocated instances to P1: 2 0 0
Allocated instances to P2: 3 0 2
Allocated instances to P3: 2 1 1
Allocated instances to P4: 0 0 2
```

```
Banker's Algorithm
1. Read Data
2. Print Data
3. Safety Sequence
4. Exit
Enter the option: 2
Pid     Alloc   Max     Need    Avail
        A B C   A B C   A B C   A B C
P0      0 1 0   7 5 3   7 4 3   3 3 2
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

Banker's Algorithm
1. Read Data
2. Print Data
3. Safety Sequence
4. Exit
Enter the option: 3
Display the Safety Sequence: P1 P3 P4 P0 P2
```

Unsafe State -

```
ks_vijay-1401@DESKTOP-J8G3TP8:~$ ./a.out

Banker's Algorithm
1. Read Data
2. Print Data
3. Safety Sequence
4. Exit
Enter the option: 1
Number of processes: 3
P0, P1, P2
Number of resources: 1
Number of Available instances of A: 1
Maximum requirement for P0: 4
Maximum requirement for P1: 5
Maximum requirement for P2: 3
Allocated instances to P0: 2
Allocated instances to P1: 3
Allocated instances to P2: 1

Banker's Algorithm
1. Read Data
2. Print Data
3. Safety Sequence
4. Exit
Enter the option: 2
Pid     Alloc   Max     Need    Avail
        A       A       A       A
P0      2       4       2       1
P1      3       5       2
P2      1       3       2
```

```
Banker's Algorithm
1. Read Data
2. Print Data
3. Safety Sequence
4. Exit
Enter the option: 3
Display the Safety Sequence: No safe sequence exists (system is in unsafe state).
```

Learning Outcomes

- Mastered Banker's algorithm for deadlock avoidance.

- Implemented resource matrices and safety checks in C.

- Developed menu-driven programs for OS simulations.

- Enhanced testing with safe/unsafe state scenarios.

- Applied deadlock prevention concepts practically.

- Gained insight into resource allocation strategies.

- Understood impact of insufficient resources via examples.