LAB REPORT

COURSE TITLE	:	Operating Systems Lab
COURSE CODE	:	CSE 210
LAB REPORT NO.	:	08
SUBMISSION DATE	:	03-11-2025

SUBMITTED TO

NAME	:	Mishal Al Rahman
DEPT. OF	:	Computer Science and Engineering (CSE) Bangladesh University of Business & Technology (BUBT)

SUBMITTED BY

NAME	:	Shadman Shahriar
ID NO.	:	20245103408
INTAKE	:	53
SECTION	:	1
PROGRAM	:	B.Sc. Engg. in CSE

Deadlocks

Task 1: Resource Allocation Graph (RAG)

Objective: Deadlock detection using a resource allocation graph for a single quantity of each resource using a cycle detection algorithm.

IDE: C/C++, Java

Since we have taken courses on **C** and **C++** in our previous semesters, I will write the algorithms in **C++**.

Deadlock detection using RAG

Here is my implementation of the deadlock detection algorithm using a **Cycle detection method**:

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

bool hasDFSCycle(int u, const vector<vector<int>> &adj,
vector<int> &color)
{
    color[u] = 1; // visiting
    for (int v : adj[u])
    {
        if (color[v] == 1)
            return true; // back to the edge → cycle
        else if (color[v] == 0)
        {
            if (hasDFSCycle(v, adj, color))
```

```
                return true;
            }
        }
        color[u] = 2; // done
        return false;
    }

    int main()
    {
        long long M, N;
        cout << "Number of Nodes: ";
        cin >> N;
        cout << "Number of Edges: ";
        cin >> M;

        vector<pair<string, string>> edges;
        edges.reserve((size_t)M);

        string a, b;
        cout << "Edges:" << endl;
        for (long long i = 0; i < M; ++i)
        {
            if (!(cin >> a >> b))
                break;
            edges.emplace_back(a, b);
        }

        unordered_map<string, int> idx;
        idx.reserve(edges.size() * 2 + 10);
        for (auto &e : edges)
        {
            if (idx.find(e.first) == idx.end())
            {
                int id = (int)idx.size();
                idx[e.first] = id;
            }
            if (idx.find(e.second) == idx.end())
            {
```

```
            int id = (int)idx.size();
            idx[e.second] = id;
        }
    }

    int n = (int)idx.size();
    vector<vector<int>> adj(n);
    for (auto &e : edges)
    {
        int u = idx[e.first];
        int v = idx[e.second];
        adj[u].push_back(v);
    }

    if (N > n)
    {
        int extra = (int)N - n;
        adj.resize((size_t)N);
        n = (int)N;
    }

    vector<int> color(n, 0);
    bool hasCycle = false;
    for (int i = 0; i < n && !hasCycle; ++i)
        if (color[i] == 0)
            if (hasDFSCycle(i, adj, color))
                hasCycle = true;

    cout << "Deadlock: " << (hasCycle ? "Yes" : "No") <<
endl;
    return 0;
}
```

Task 2: Banker's Algorithm

Objective: Simulate Banker's Algorithm used for Deadlock Avoidance and find whether the system is in safe state or not.

IDE: C/C++, Java

Here is my implementation of the banker's algorithm in **C++**:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int m, n;
    cout << "Enter the no. of processes: ";
    cin >> n;
    cout << "Enter the no. of resources: ";
    cin >> m;

    vector<vector<int>> maxNeed(n, vector<int>(m));
    vector<vector<int>> alloc(n, vector<int>(m));
    vector<vector<int>> need(n, vector<int>(m));
    vector<int> avail(m);
    vector<int> total(m);
    vector<int> finish(n, 0);
    vector<int> safeSequence;

    for (int i = 0; i < n; i++)
    {
        cout << endl;
        cout << "Process " << i + 1 << endl;
        for (int j = 0; j < m; j++)
        {
            cout << "Maximum value for resource " << j + 1
```

```
<< ": ";
            cin >> maxNeed[i][j];
        }
        for (int j = 0; j < m; j++)
        {
            cout << "Allocated from resource " << j + 1 <<
": ";
            cin >> alloc[i][j];
        }
    }

    cout << endl;
    for (int j = 0; j < m; j++)
    {
        cout << "Enter total value of resource " << j + 1
<< ": ";
        cin >> total[j];
    }

    for (int k = 0; k < m; k++)
    {
        int totAlloc = 0;
        for (int l = 0; l < n; l++)
            totAlloc += alloc[l][k];
        avail[k] = total[k] - totAlloc;
    }

    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = maxNeed[i][j] - alloc[i][j];

    cout << endl;

    int count = 0;
    while (count < n)
    {
        int found = 0;
        for (int i = 0; i < n; i++)
```

```
        {
            if (!finish[i])
            {
                int canFinish = 1;
                for (int j = 0; j < m; j++)
                {
                    if (need[i][j] > avail[j])
                    {
                        canFinish = 0;
                        break;
                    }
                }

                if (canFinish)
                {
                    for (int k = 0; k < m; k++)
                        avail[k] += alloc[i][k];
                    safeSequence.push_back(i);
                    finish[i] = 1;
                    found = 1;
                    count++;
                }
            }
        }

        if (!found)
        {
            cout << "The system is not in a safe state.";
            return 0;
        }
    }

    cout << "The system is currently in safe state and ";
    cout << "< ";
    for (int i = 0, z = safeSequence.size(); i < z; i++)
        cout << "P" << safeSequence[i] + 1 << " ";
    cout << "> is the safe sequence.\n";
}
```

Code Analysis of the Banker's Algorithm

In the Banker's algorithm, the process and resource data is stored in a `n × m` matrix. We are using `Vector` s instead of raw arrays for convenience.

Structure	Description
`avai[m]`	Available resources of each type.
`max[n][m]`	Maximum demand of each process.
`alloc[n][m]`	Resources currently allocated.
`need[n][m]`	Remaining resources needed. (`max - alloc`)

Here is how the Banker's algorithm takes decisions:

1. **Initialization**
    - work = available
    - `finish[i] = false` for all processes
2. **Find a Process**
   Where,
    - `need[i] ⩽ work` and
    - `finish[i] == false`
3. **Simulate Completion**
    - Allocate and then release its resources:
        - `work = work + alloc[i]`

- Mark process finished.

4. **Repeat**
   Continue until,

   - all processes finish (safe state) or
   - none can proceed (unsafe)

5. **Decision**

   - If all processes finish: safe state
   - Else: unsafe (possible deadlock)

Here is how total allocated and need matrices are calculated:

```
for (int k = 0; k < m; k++)
{
    int totAlloc = 0;
    for (int l = 0; l < n; l++)
        totAlloc += alloc[l][k];
    avail[k] = total[k] - totAlloc;
}

for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        need[i][j] = maxNeed[i][j] - alloc[i][j];
```

Here is how the safety algorithm works:

```
int count = 0;
while (count < n) {
    int found = 0;
```

```
    // ...
}
```

- `count` tracks how many processes have successfully completed
- `found` tracks if at least one process could safely finish in this iteration.
- The loop continues until all `n` processes have finished.

```
for (int i = 0; i < n; i++) {
    if (!finish[i]) {
        // ...
    }
    // ...
}
```

- It iterates through each process `i`.
- Initially `finish[i] == 0`, meaning the process has not yet completed.

Now,

```
int canFinish = 1;
for (int j = 0; j < m; j++) {
    if (need[i][j] > avail[j]) {
        canFinish = 0;
        break;
    }
}
```

- For every resource type `j`, check if the remaining need of process i (`need[i][j]`) can be satisfied by the currently available resources (`avail[j]`).

- If even one resource type is insufficient, `canFinish = 0`

If the process can finish,

```
if (canFinish) {
    for (int k = 0; k < m; k++)
        avail[k] += alloc[i][k]; // release its allocated
resources

    safeSeq.push_back(i);         // record process in safe
order
    finish[i] = 1;                // mark as completed
    found = 1;                    // at least one process
could run
    count++;                      // increase no. of
finished processes
}
```

If after checking all processes, none can safely proceed (`found == 0`), the system is supposed to be in an unsafe state.

But if the `count < n` loop completes successfully, it means,

- Every process could finish eventually.
- The system is safe, and the safe sequence of execution is printed.

Unlike the RAG method, the Banker's algorithm is also preventive. It checks whether allocating resources to a process keeps the system in a **safe state**. (A **safe state** means there exists **at least one safe sequence** in which all processes can complete.)