Lab Project

CS-303 Operating System

Submitted to:

Mam Syeda Aimal Fatima Naqvi

Submitted By:

Ayesha Ikram	23-CyS-019
Eelaf Khan	23-CyS-029
Zoha Munawar	23-CyS-061

Dated:

June 16th, 2025

Department of Cyber Security

HITEC University, Taxila

Operating System Simulation Project Report

1. Project Objective:

The objective of this project is to design and implement a dynamic, multi-functional Operating System simulation in C++ that demonstrates core OS concepts including:

- Process Management
- Memory Management
- CPU Scheduling Algorithms (FCFS and Round Robin)
- Deadlock Detection and Avoidance (Banker's Algorithm)
- Inter-Process Communication (Producer-Consumer Problem)
- System Performance Analysis

➢ This project aims to provide hands-on understanding and practical application of theoretical OS concepts within the domain of Cyber Security.

2. Project Scope:

- Dynamic user inputs (no hardcoding)
- Scheduling using FCFS and Round Robin
- Memory management simulation
- Deadlock detection and avoidance using Banker's Algorithm
- Producer-Consumer simulation for inter-process communication.
- Simple and understandable code structure

3. Methodology and Design:

3.1 Programming Language:

- Language: C++

3.2 Design Features:

- Modular functions for each OS concept
- User-friendly input prompts
- Use of standard data structures like vectors and queues

3.3 Modules Description:

a. Process Scheduling (FCFS and Round Robin):

- FCFS: Non-preemptive, processes sorted by arrival time.
- Round Robin: Preemptive, time quantum-based CPU sharing.

b. Memory Management:

- Allocates memory to processes until system memory is exhausted.

c. Deadlock Detection and Avoidance:

- Deadlock detection using Banker's Safety Algorithm.
- Avoidance through Banker's Algorithm to maintain safe state.

d. Inter-Process Communication:

- Simulates Producer-Consumer Problem using queue buffer.

4. Implementation Details:

4.1 Inputs Taken from User:

- Number of processes
- Process arrival time, burst time, memory requirement
- Total system memory
- Time quantum for Round Robin
- Allocation, Max Demand, and Available matrices for Deadlock Detection
- Number of items for Producer-Consumer problem

4.2 Code Modules Overview:

- **FCFS Scheduling:** Simple execution based on arrival.
- **Round Robin Scheduling:** Executes based on user-defined quantum.
- **Memory Management:** Allocates memory; displays if insufficient.
- **Deadlock Detection:** Uses Safety Algorithm to detect unsafe states.
- **Deadlock Avoidance:** Banker's Algorithm determines safe sequences.
- **Producer-Consumer:** Simulates bounded buffer (size = 5).

5. Code:

- Firstly, I create a file which name is misty.cpp

```
ubuntu@ubuntu:~$ touch misty.cpp
ubuntu@ubuntu:~$ ls
aez      Desktop    Downloads  Music  OS      OS.cpp    Public  Templates
aez.cpp  Documents  misty.cpp  os     os.cpp  Pictures  snap    Videos
ubuntu@ubuntu:~$ nano misty.cpp
```

- Then I write a Code in that file.

```
 GNU nano 7.2                                misty.cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

// Process structure
struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int remainingTime;
};

// FCFS Scheduling
void FCFS(vector<Process> processes) {
    sort(processes.begin(), processes.end(), [](Process a, Process b) {
        return a.arrivalTime < b.arrivalTime;
    });

    int time = 0;
    cout << "\n--- FCFS Scheduling ---\n";
    for (auto p : processes) {
        if (time < p.arrivalTime)
            time = p.arrivalTime;
        cout << "Process " << p.pid << " executing from " << time << " to " << time + p.burstTime << endl;
        time += p.burstTime;
    }
}

// Round Robin Scheduling
void RoundRobin(vector<Process> processes, int quantum) {
    cout << "\n--- Round Robin Scheduling ---\n";
    queue<Process> q;
    int time = 0;

    for (auto p : processes)
        q.push(p);

    while (!q.empty()) {
        Process p = q.front();
        q.pop();

        if (p.arrivalTime > time)
            time = p.arrivalTime;

        int execTime = min(quantum, p.remainingTime);
        cout << "Process " << p.pid << " executed for " << execTime << " units." << endl;
        time += execTime;
        p.remainingTime -= execTime;

        if (p.remainingTime > 0)
            q.push(p);
    }
}
```

```cpp
// Memory Management Simulation
void memoryManagementSimulation(int totalMemory, vector<int> processMemory) {
    cout << "\n--- Memory Management ---\n";
    int allocated = 0;

    for (int i = 0; i < processMemory.size(); i++) {
        if (allocated + processMemory[i] <= totalMemory) {
            cout << "Process " << i + 1 << " allocated " << processMemory[i] << " MB\n";
            allocated += processMemory[i];
        } else {
            cout << "Not enough memory for Process " << i + 1 << endl;
        }
    }
}

// Deadlock Detection
void deadlockDetection(int p, int r, vector<vector<int>> alloc, vector<vector<int>> max, vector<int> avail) {
    vector<int> finish(p, 0);
    vector<int> work = avail;

    cout << "\n--- Deadlock Detection ---\n";
    bool done;

    do {
        done = false;
        for (int i = 0; i < p; i++) {
            if (!finish[i]) {
                bool canFinish = true;
                for (int j = 0; j < r; j++) {
                    if (max[i][j] - alloc[i][j] > work[j]) {
                        canFinish = false;
                        break;
                    }
                }
                if (canFinish) {
                    for (int j = 0; j < r; j++) work[j] += alloc[i][j];
                    finish[i] = 1;
                    done = true;
                }
            }
        }
    } while (done);

    for (int i = 0; i < p; i++) {
        if (!finish[i]) {
            cout << "Deadlock detected.\n";
            return;
        }
    }
    cout << "No Deadlock detected.\n";
}

// Banker's Algorithm
bool isSafe(int p, int r, vector<vector<int>> alloc, vector<vector<int>> max, vector<int> avail) {
```

```
    vector<int> work = avail;
    vector<bool> finish(p, false);
    cout << "\n--- Banker's Algorithm (Deadlock Avoidance) ---\n";

    for (int count = 0; count < p; count++) {
        bool found = false;
        for (int i = 0; i < p; i++) {
            if (!finish[i]) {
                bool possible = true;
                for (int j = 0; j < r; j++) {
                    if (max[i][j] - alloc[i][j] > work[j]) {
                        possible = false;
                        break;
                    }
                }
                if (possible) {
                    for (int j = 0; j < r; j++) work[j] += alloc[i][j];
                    finish[i] = true;
                    found = true;
                }
            }
        }
        if (!found) return false;
    }
    return true;
}

// Producer-Consumer Problem (Simulation)
void producerConsumerSimulation() {
    cout << "\n--- Producer-Consumer Simulation ---\n";
    queue<int> buffer;
    int bufferSize = 5;
    int items;

    cout << "How many items to produce? ";
    cin >> items;

    for (int i = 1; i <= items; i++) {
        if (buffer.size() < bufferSize) {
            buffer.push(i);
            cout << "Produced item " << i << endl;
        } else {
            cout << "Buffer full! Producer waiting...\n";
        }
    }

    while (!buffer.empty()) {
        cout << "Consumed item " << buffer.front() << endl;
        buffer.pop();
    }
}

// Main function
int main() {
    int n, totalMemory, quantum;
    cout << "Enter number of processes: ";
```

```
    cin >> n;

    vector<Process> processes(n);
    vector<int> processMemory(n);

    // Input for processes
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Process " << i + 1 << " Arrival Time: ";
        cin >> processes[i].arrivalTime;
        cout << "Process " << i + 1 << " Burst Time: ";
        cin >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;

        cout << "Process " << i + 1 << " Memory Requirement: ";
        cin >> processMemory[i];
    }

    cout << "Enter total system memory: ";
    cin >> totalMemory;

    FCFS(processes);

    cout << "Enter Time Quantum for Round Robin: ";
    cin >> quantum;
    RoundRobin(processes, quantum);

    memoryManagementSimulation(totalMemory, processMemory);
```

```
    // Deadlock detection inputs
    int p, r;
    cout << "\n--- Deadlock Detection & Avoidance Inputs ---\n";
    cout << "Enter number of processes: ";
    cin >> p;
    cout << "Enter number of resource types: ";
    cin >> r;

    vector<vector<int>> alloc(p, vector<int>(r));
    vector<vector<int>> max(p, vector<int>(r));
    vector<int> avail(r);

    cout << "Enter Allocation Matrix:\n";
    for (int i = 0; i < p; i++)
        for (int j = 0; j < r; j++)
            cin >> alloc[i][j];

    cout << "Enter Maximum Demand Matrix:\n";
    for (int i = 0; i < p; i++)
        for (int j = 0; j < r; j++)
            cin >> max[i][j];

    cout << "Enter Available Resources:\n";
    for (int i = 0; i < r; i++) cin >> avail[i];

    deadlockDetection(p, r, alloc, max, avail);

    if (isSafe(p, r, alloc, max, avail))
```

```
        cout << "The system is in a safe state.\n";
    else
        cout << "The system is not in a safe state (Deadlock possible).\n";

    producerConsumerSimulation();

    return 0;
}
```

6. Explanation & Summary of Implemented Modules:

This project is a complete Operating System Simulation made in C++. It explains important concepts like Process Scheduling, Memory Management, Deadlock Detection and Avoidance, and Inter-Process Communication (IPC). The system is fully dynamic, meaning it asks the user for all inputs instead of using fixed or hardcoded values.

6.1. Process Scheduling:

a. First-Come, First-Served (FCFS) Scheduling:

- Non-preemptive scheduling technique.
- Processes are executed according to their **arrival time**.
- Each process runs until it is fully completed.
- Demonstrates simple CPU scheduling and the possibility of process starvation.

b. Round Robin (RR) Scheduling:

- Preemptive scheduling with **user-defined time quantum**.
- Each process gets equal CPU time in **cycles/rounds**.
- If not finished within the quantum, the process returns to the queue.
- Ensures fair CPU usage and reduces process waiting time.

6.2. Memory Management Simulation:

- Simulates **dynamic memory allocation** to each process.
- Each process requests a specific amount of memory.
- The system checks:
    - If memory is available → **memory allocated**.
    - If not → displays **"Not enough memory"**.
- Shows how real operating systems manage limited memory resources.

6.3. Deadlock Handling:

a. Deadlock Detection (Safety Algorithm):

- Detects if the system is in a **deadlock state**.
- Uses user-input matrices:
  - **Allocation matrix**.
  - **Maximum demand matrix**.
  - **Available resources**.
- Declares deadlock if any process cannot finish because required resources are unavailable.

b. Deadlock Avoidance (Banker's Algorithm):

- Checks if the system can safely allocate resources to all processes.
- Determines if a **safe sequence of process execution exists**.
- If no safe sequence → system warns about a possible unsafe (deadlock) state.

6.4. Inter-Process Communication (Producer-Consumer Simulation):

- Simulates the Producer-Consumer Problem using **a** bounded buffer (size 5)**.**
- **Producer:**
  - Produces items and places them into the buffer.
  - Waits if the buffer is full.
- **Consumer:**
  - Removes and processes items from the buffer.
- Demonstrates process synchronization and shared resource handling, as in real operating systems.

6.5. User Inputs Collected:

- Number of processes.
- For each process:
  - Arrival time.
  - Burst time.
  - Memory requirement.
- Total system memory.
- Time quantum (for Round Robin Scheduling).
- Deadlock Handling:
  - Number of processes and resource types.
  - Allocation matrix.
  - Maximum demand matrix.
  - Available resources.
- Number of items for Producer-Consumer problem.

7. Output:

- I run the commands to get output:

```
ubuntu@ubuntu:~$ g++ misty.cpp -o misty
ubuntu@ubuntu:~$ ./misty
```

- And the Output is:
  - I entered all the data which is required:

```
Enter number of processes: 3
Process 1 Arrival Time: 0
Process 1 Burst Time: 4
Process 1 Memory Requirement: 10
Process 2 Arrival Time: 1
Process 2 Burst Time: 3
Process 2 Memory Requirement: 20
Process 3 Arrival Time: 2
Process 3 Burst Time: 2
Process 3 Memory Requirement: 30
Enter total system memory: 50
```

  - FCFS Scheduling:

```
--- FCFS Scheduling ---
Process 1 executing from 0 to 4
Process 2 executing from 4 to 7
Process 3 executing from 7 to 9
```

  - Round Robin Scheduling:

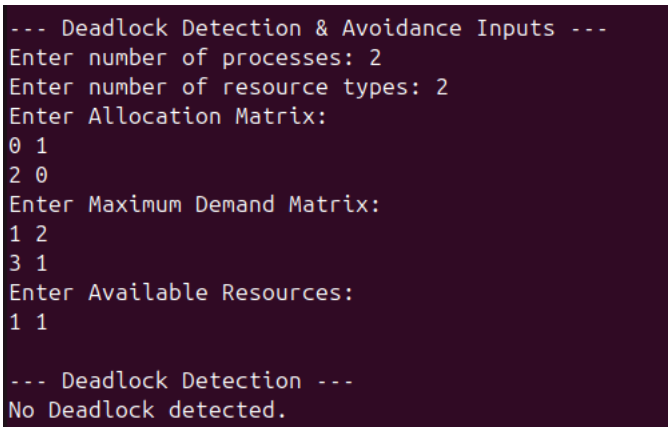
```
Enter Time Quantum for Round Robin: 2

--- Round Robin Scheduling ---
Process 1 executed for 2 units.
Process 2 executed for 2 units.
Process 3 executed for 2 units.
Process 1 executed for 2 units.
Process 2 executed for 1 units.
```

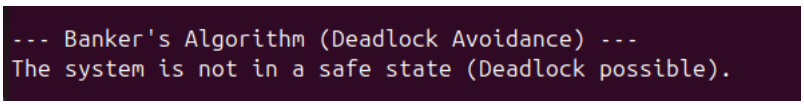
  - Memory Management Simulation:

```
--- Memory Management ---
Process 1 allocated 10 MB
Process 2 allocated 20 MB
Not enough memory for Process 3
```

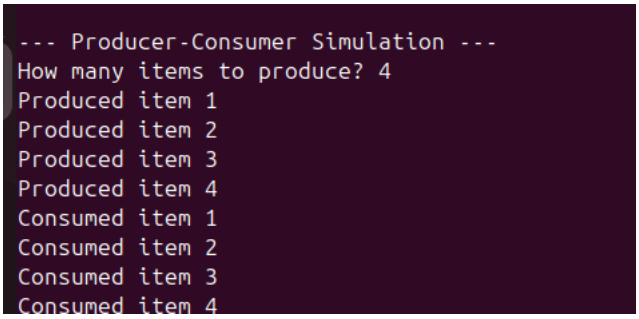
- Deadlock Detection:



- Banker's Algorithm:



- Producer-Consumer Problem:



8. Performance Measures and Comparative Analysis:

Technique	Description	Performance Observation
FCFS	Simple & easy, non-preemptive	Possible process starvation
Round Robin	Fair sharing with time quantum	Reduced waiting time, better CPU utilization
Deadlock Detection	Banker's Algorithm (Safety Check)	Detects potential unsafe states
Deadlock Avoidance	Banker's Algorithm	Prevents unsafe resource allocation
Producer-Consumer	Queue-based buffer problem simulation	Synchronization problem simulated

9. Conclusion:

This OS simulation successfully demonstrates key concepts of Operating System including CPU Scheduling, Memory Management, Deadlock Detection/Avoidance, and Inter-Process Communication. The interactive design allows users to understand how system resources are managed dynamically. The system ensures fairness, avoids deadlocks, and simulates real-world synchronization scenarios.

10. Project Evaluation:

- Effective scheduling can greatly influence system performance.
- Deadlock handling is crucial for safe OS operations.
- IPC problems like Producer-Consumer are essential in real OS design.