Complex Computing Problem (CCP)

Title

Design and Implementation of a Mini Operating System Simulator
(Expandable to Final Year Project Scope)

WEIGHTAGE

10 Marks (Absolute)
Time Allowed: 2 Weeks
Course: Operating Systems Lab (Undergraduate)

EDUCATIONAL INTENT

This CCP is designed to help you:

- Apply **core OS concepts together**, not in isolation
- Build a **small but realistic simulator**, not a toy program
- Lay a **foundation that can be expanded into a Final Year Project (FYP)** such as a cloud OS simulator, kernel-level scheduler, or distributed resource manager

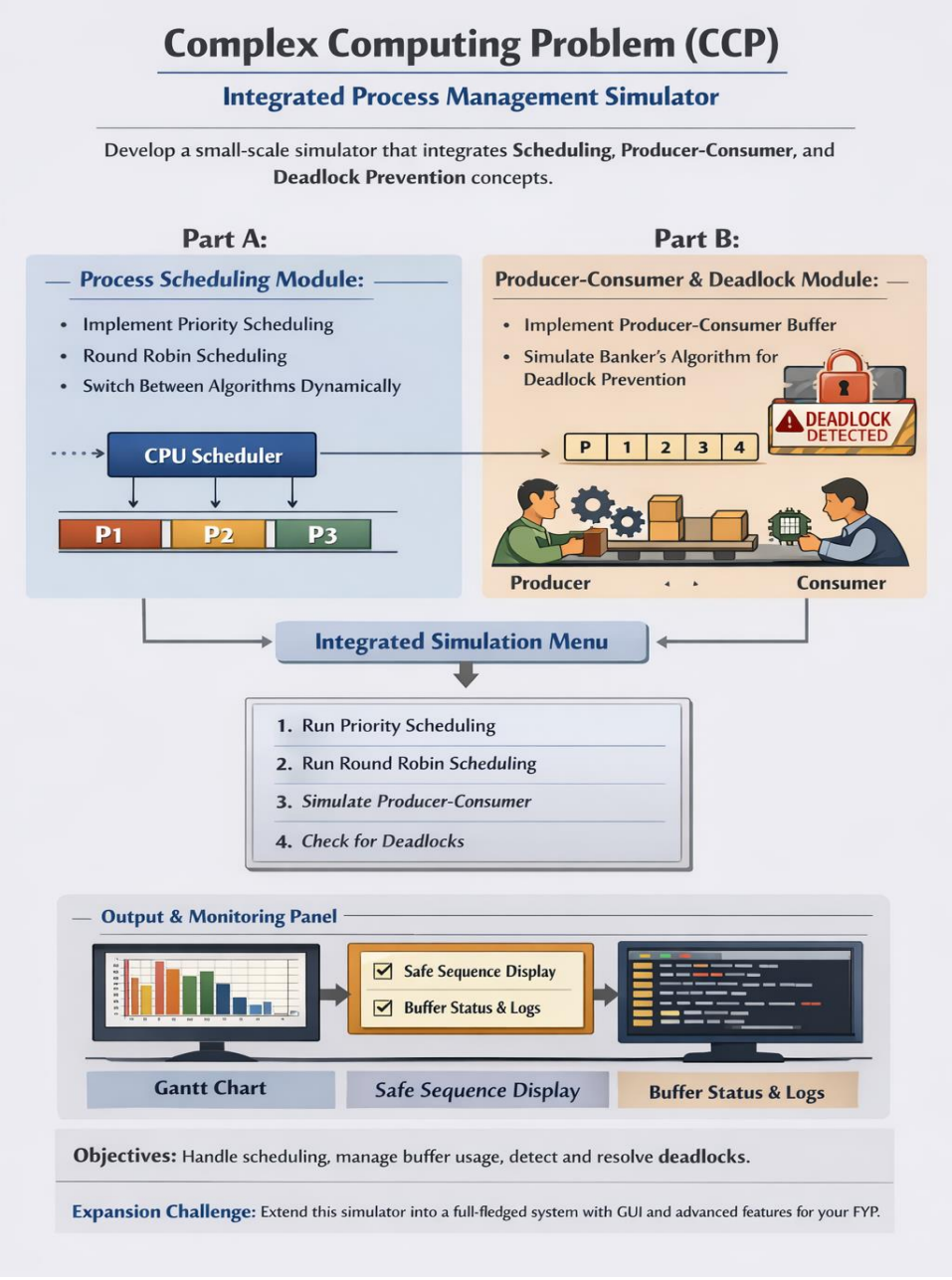
You are NOT expected to build a real OS. You are expected to **think like an OS designer**.

PROBLEM STATEMENT

Design and implement a **command-line based Mini OS Simulator** that models how an operating system:

1. Schedules processes
2. Handles concurrent production and consumption of tasks
3. Prevents unsafe resource allocation (deadlock)

The simulator must run as a **single integrated system**.



SYSTEM OVERVIEW (WHAT YOU ARE BUILDING)

Your simulator will consist of **three cooperating modules**:

PART A — PROCESS & SCHEDULER MODULE (40%)

OBJECTIVE

Simulate **CPU scheduling** using two classical algorithms.

MANDATORY FEATURES

1. Each process must have:
    - Process ID
    - Arrival Time
    - CPU Burst Time
    - Priority
    - Resource Requirement Vector (R1, R2, …)
2. Implement **both**:
    - Priority Scheduling (Non-preemptive)
    - Round Robin Scheduling (Preemptive)
3. Scheduler Selection Rule (Fixed for CCP):

Condition	Scheduler Used
≤ 5 ready processes	Priority Scheduling
> 5 ready processes	Round Robin

4. Display after execution:
    - Gantt Chart (text-based)
    - Waiting Time
    - Turnaround Time
    - Average statistics
    -

PART B — PRODUCER–CONSUMER MODULE (30%)

OBJECTIVE

Simulate how processes enter the system concurrently.

REQUIREMENTS

1. Implement:
    - At least **2 producer threads**
    - At least **1 consumer thread** (CPU)
2. Producers:
    - Generate processes dynamically
    - Insert them into a **bounded buffer (ready queue)**
3. Consumer:
    - Fetches processes from the buffer
    - Sends them to the scheduler
4. Synchronization:
    - Use **semaphores** for empty/full buffer
    - Use **mutex** for mutual exclusion
      Please install clang-20 and build-essential tools before compiling semaphore.

      sudo apt install build-essential g++ -y
      sudo apt install clang-20 -y

⚠ Busy waiting is strictly forbidden.

PART C — RESOURCE MANAGEMENT & DEADLOCK PREVENTION (30%)

OBJECTIVE

Ensure the system never enters an unsafe state.

REQUIREMENTS

1. Implement a **simplified Banker's Algorithm**:
    - Fixed number of resource types
    - Max, Allocation, Available matrices
2. Before a process executes:
    - Check if granting its resources keeps system safe
3. If unsafe:
    - Process must be **blocked and queued**
4. Display:
    - Safe sequence (if exists)
    - Blocked processes list

USER INTERACTION (SIMPLE & CLEAR)

Your program must provide a **menu-based interface**:

1. Start Simulation

2. Add Process Manually

3. Display System State

4. Exit

No GUI required.

CONSTRAINTS (VERY IMPORTANT)

✔ Use C / C++ (with proper threading)

✔ Code must be modular

✘ No hardcoded scheduling decisions

✘ No infinite loops without explanation

✘ No single-file monolith solutions

DELIVERABLES

1. SOURCE CODE

- Properly structured
- Well-commented
- Uploaded to GitHub

2. PDF REPORT (MAX 8 PAGES)

Must include:

1. Module-wise explanation
2. Scheduling decisions
3. Semaphore usage explanation
4. Deadlock prevention logic
5. Limitations of current simulator

MARKING RUBRIC FOR VIVA

Component	Marks

Scheduler correctness	3
Synchronization logic	2
Deadlock prevention	2
Integration	2
Report quality	1

HOW THIS CCP EXTENDS TO FYP (IMPORTANT)

Students may later expand this into:

- GUI-based OS Simulator
- Cloud workload scheduler
- Distributed resource manager
- Container orchestration simulator
- Real-time OS scheduler

This CCP is your **foundation**, not your limit.

FINAL NOTE

This CCP rewards:

- Understanding
- Design clarity
- Correct synchronization

It punishes:

- Blind coding
- AI dependency
- Memorized solutions

If you understand every line you write, you will score full marks.