# Operating Systems
# CS-2006 (Spring 25)
# Assignment-02

**Submission Date: 30 March 2025**

---

## Instructions:

- *This is an **individual assignment**.*
- *All parties involved in any kind of **plagiarism/cheating** (even in a single line of code) will receive **zero marks** in all assignments.*
- ***Assignment deadline won't be extended**.*
- ***Late submissions will be discarded**, so submit your assignment on time.*

## Submission Guidelines:

*You must follow the guidelines below; otherwise, your submission **won't be accepted**:*

1. ***Create a new directory**.*
2. ***Rename the directory** in the following format:*
   ***YOURSECTION_ROLLNUMBER_NAME***
   - *Example: `A_22I-1183_Faisal Jabbar`*
3. ***Put all your files** (`.cpp`, .c and `.txt` **only**) into this newly created directory.*
4. ***Compress the directory** into a `.zip` file.*
5. ***Submit it on Google Classroom**.*

## Additional Requirements:

- *Use **good programming practices**:*
  - *Well-commented and properly indented code.*
  - *Meaningful variable names.*
  - *Readable and structured code.*
- ***Understanding the problem is also part of the assignment.***

## Question 1:(50 Marks)

# Multiplayer Terminal Game - Zombie Survival

Your task is to implement a multiplayer zombie survival game in the terminal using process management techniques such as fork(), exec(), and pipes(). This game will simulate players working together (or betraying each other) to fight off waves of zombies.

**Additionally, you will containerize your game using Docker.**

Each student must introduce a unique element to their game by modifying the number of zombie processes based on their roll number (see **Unique Student Requirement** section).

### Game Concept

- The game manager (parent process) controls the overall game logic.
- Players are child processes that send actions (move, attack, hide) through pipes.
- Zombies are separate processes that move towards players and attack them.
- The game runs in turns, updating the state based on player and zombie actions.

### Gameplay Flow

1. **Game Start**
    - Players join as forked child processes.
    - Zombies are spawned dynamically as separate processes.
    - The parent process manages all entities.
2. **Player Actions (Sent via Pipes to Game Manager)**
    - Move: Change position (left, right, up, down).
    - Attack: Hit a zombie within range.
    - Hide: Reduce the chance of being attacked.
    - Use Items: Health kits, weapons, etc.
3. **Zombie AI (Separate Processes Communicating via Pipes)**
    - Zombies move towards the nearest player.
    - If they reach a player, they attack.
    - Dead zombies are removed from the game.
4. **Win/Loss Conditions**
    - **Players Win** if they eliminate all zombies within the given waves.
    - **Zombies Win** if all players die.

**Technical Implementation**

1. **Process Handling**
    - The game manager (parent process) forks child processes for:
        - Players
        - Zombies
    - Pipes facilitate communication between players, zombies, and the game manager.
2. **Inter-Process Communication (Pipes)**
    - Players send actions (attack, move, hide) via pipes.
    - Zombies send position and attack updates via another pipe.
    - The game manager updates and prints the battlefield state.
3. **Using exec() for Special Actions**
    - Certain actions can trigger external scripts using exec(), such as loot generation.
    - Example:

execvp("python3", ["python3", "generate_loot.py", NULL]);

- This spawns an external script to generate random loot items.

## Game Example (Terminal Output)

```
====== ZOMBIE SURVIVAL GAME ======
Player 1: [Health: 100]  [Weapon: Knife]
Player 2: [Health: 80]   [Weapon: Pistol]
Zombies: 3 approaching...

>> Player 1 attacks Zombie 1! [Zombie 1 Health: 50]
>> Zombie 2 bites Player 2! [Player 2 Health: 60]
>> Player 2 finds a shotgun! [Damage Increased]

[Next Turn]
```

**Unique Student Requirement**

Each student must determine the number of zombie processes dynamically using their roll number.

1. Take the last four digits of your roll number (XXXX).
2. Multiply it by any constant number greater than 2.
3. Take the result modulus 20 ((XXXX * 3) % 20).
4. The final result is the number of zombies in the game.

**Submission Requirements**

- Source code (.c and any scripts used)
- Dockerfile for containerization
- Documentation (README.md) explaining:
    - How to compile and run the game (pull link and instructions)
    - How the pipes and process management are implemented
    - How the unique student requirement is implemented
- Screenshot of terminal output running the game

**Extensions & Enhancements (Optional)**

- Introduce different zombie types (fast zombies, boss zombies, etc.)
- Implement crafting mechanics for survival elements.
- Add a leaderboard to track the best survivors.

## Question 02:(50 Marks)

# Synchronizing Professor and Students During Office Hours

You have been hired by the Computer Science Division to develop a thread-based system that coordinates a professor and students during office hours.

During office hours:

- The **professor takes a nap** if no students are waiting to ask questions.
- When a student arrives, they must ensure that:
    1. **Only one student asks a question at a time.**
    2. **Each question is answered before another question can be asked.**
    3. **Students wait until the professor finishes answering before asking another question.**

## Function Requirements

You must implement the following four functions:

1. **QuestionStart()** – Called when a student arrives to ask a question. It should block until it is the student's turn.
2. **QuestionDone()** – Called after the student has asked the question. It should ensure the student does not ask another question until the professor responds.
3. **AnswerStart()** – Called by the professor to start answering. It should block until a student has asked a question.
4. **AnswerDone()** – Called after the professor finishes answering, allowing the next student to ask.

## Constraints

1) **Thread synchronization must be achieved WITHOUT using mutexes or Semaphores(Locks).**
2) The solution must avoid **busy waiting** or inefficient polling.

Your task is to implement these four functions to correctly synchronize the interactions between the professor and students.

## Question 03:(40 Marks)

# Cricket Match Analysis Tool with Scheduling Algorithms

**Objective:** Develop a tool in C that utilizes CPU scheduling algorithms (SJF, SRTF, and Round Robin) to optimize and analyze the batting and bowling order for cricket matches, aimed at enhancing team performance based on historical data.

**Background:** In cricket, the order of batsmen and bowlers can significantly impact the outcome of the game. Efficient management of player order based on their performance stats (like batting average and bowling economy) and current form can lead to better strategies and game outcomes.

**Problem Statement:** Your task is to create a program that simulates different arrangements of batsmen and bowlers using various CPU scheduling algorithms to determine the most effective lineup for given match scenarios.

**Inputs:**

1. Number of players.
2. Players' statistics: Batting average, strike rate, bowling average, and economy rate.
3. Historical performance data (e.g., performance under different match conditions).
4. Player current form index (a metric that combines recent performance with historical data).

**Requirements:**

1. Implement SJF to prioritize batsmen with higher strike rates and bowlers with lower economy rates.
2. Use SRTF to dynamically adjust the order based on ongoing simulations of match scenarios, considering players' current form.
3. Round Robin could be used to simulate a rotation policy for bowlers to manage their workload and effectiveness over a series.
4. Your program should simulate these algorithms and produce a report that outlines:
    - Optimal batting order.
    - Optimal bowling sequence.
    - Predicted match outcomes based on historical and current form data.

**Expected Outputs:**

- A detailed analysis report that includes:
    - Predicted run scores and wickets for each batting and bowling order.
    - Comparisons of each scheduling strategy's effectiveness in maximizing runs scored and minimizing runs conceded.
    - Recommendations for the best batting and bowling orders for upcoming matches.

**Challenges:**

- Integrating complex cricket statistics effectively into scheduling algorithms.
- Handling variability in player performance due to external factors like match conditions and opposition strengths.
- Providing a user-friendly output that clearly shows the advantages of one scheduling strategy over another.

---

## Question 04 (40 Marks)

# Four-Process Circular Communication System with File Logging Using Unnamed Pipes

### Problem Statement:

Develop a **C program** where **four processes communicate** in a **circular pattern** using **unnamed pipes**. Each process should **modify incoming messages**, log them to a file, and **pass them to the next process**.

### Processes Setup:

- **Process A:**
    - Initiates the message cycle.
    - Logs messages to a file (`log_A.txt`).
    - Sends the message to **Process B**.
- **Process B:**
    - Receives the message from **Process A**.
    - Appends `" - Processed by B"`.
    - Logs the modified message to `log_B.txt`.
    - Sends the message to **Process C**.
- **Process C:**

    - Receives the message from **Process B**.
    - Appends `" - Processed by C"`.
    - Logs the modified message to `log_C.txt`.
    - Sends the message to **Process D**.
- **Process D:**
    - Receives the message from **Process C**.
    - Appends `" - Processed by D"`.
    - Logs the modified message to `log_D.txt`.
    - Sends the message **back to Process A**.

## File Handling Requirement:

- Each process **must create its own log file** (e.g., `log_A.txt`, `log_B.txt`, etc.).
- Each process **writes the modified message** to its respective log file **before passing it to the next process**.
- **System calls** should be used to open, write, and close files.

## Requirements:

- Use `fork()` to create **four separate processes**.
- Establish **unnamed pipes** for circular **message passing** between processes.
- Implement **file operations** within each process to **log messages** as they are processed.
- Ensure **robust error handling** for both **IPC (Inter-Process Communication) and file operations**.
- Handle **synchronization of message passing** to maintain **order and completeness of data**.

## Expected Outputs:

- **Each process** should maintain **a log file** (`log_A.txt`, `log_B.txt`, etc.) with **all messages it has processed**.
- **Process A** should **display** each **fully modified message** after receiving it back from **Process D**.

---

## Question 05 (20 Marks)

You are a **system designer** tasked with implementing a **3-Level Multi-Feedback Queue (MFQ) CPU Scheduling Algorithm** in **C/C++**. The system works as follows:

**Queue 1 (Highest Priority):**

- Uses **Round Robin (RR)** scheduling with a **time quantum of 4**.
- If a process is not completed within this time, it moves to **Queue 2**.

**Queue 2 (Medium Priority):**

- Uses **Round Robin (RR)** scheduling with a **time quantum of 8**.
- If a process is still not completed, it moves to **Queue 3**.

**Queue 3 (Lowest Priority):**

- Uses **First-Come, First-Serve (FCFS)** scheduling.
- Processes in this queue run **until completion**.

## Requirements:

Write a **C/C++ program** that simulates **MFQ scheduling** for a set of processes. Each

process has:

- **A Process ID (PID)**
- **Arrival Time** (when it enters the system)
- **Burst Time** (total CPU execution time required)

Your program should **display** the execution order of processes and calculate:

- **Turnaround Time (TAT) = Completion Time - Arrival Time**
- **Waiting Time (WT) = Turnaround Time - Burst Time**

## Input Format:

Your program should take **N (number of processes)** as input. Each process should have the following details:

```
Process ID, Arrival Time, Burst Time
```

**Example Input:**

```
4
1 0 20
2 2 10
3 4 15
4 6 12
```

## Expected Output Format:

The program should print:

```
Statistics:
Process 1 -> Turnaround Time: X, Waiting Time: Y
Process 2 -> Turnaround Time: X, Waiting Time: Y
Process 3 -> Turnaround Time: X, Waiting Time: Y
Process 4 -> Turnaround Time: X, Waiting Time: Y
```

**Happy Coding!** 😊👨‍💻