PREMIER UNIVERSITY CHATTOGRAM

DEPARTMENT OF **COMPUTER SCIENCE AND ENGINEERING**

ASSIGNMENT

COURSE NAME	Operating Systems
COURSE CODE	CSE 3733
ASSIGNMENT TOPIC	1: Designing a Simplified Operating System for 3 different scenario. 2: Page Replacement Algorithm Analysis.
DATE OF ASSIGNMENT	07th February 2025

SUBMITTED TO

SHATABDI ROY MOON

LECTURER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REMARKS	**SUBMITTED BY**	
	NAME	Rimjhim Dey
	ID	0222220005101039
	SEMESTER	5th
	BATCH	42nd
	SESSION	Fall 2024
	SECTION	A

Answer to Question No. 1

Scenario 1: A real-time operating system for a drone fleet managing autonomous flight.

Overview:

In the context of managing a drone fleet for autonomous flight, a Real-Time Operating System (RTOS) becomes crucial to ensure all drones in the fleet work cohesively. The RTOS is designed to handle the coordinated, time-sensitive tasks necessary for drones to operate in unison for complex missions. The primary purpose of this RTOS is to ensure seamless task scheduling, real-time data processing, and communication between drones, while maintaining strict safety protocols and power efficiency.

Key Features and Architecture for RTOS in Autonomous Drone Fleet Management:

1.Microkernel Architecture

- **Lightweight Microkernel**: A microkernel ensures that the RTOS remains lightweight, with minimal overhead, which is crucial in environments with limited hardware resources (like drones). It supports only essential functionalities such as task scheduling, inter-process communication (IPC), and hardware management.
- **Minimal Overhead**: Ensures that the RTOS can handle real-time data processing from multiple drones without taxing the system.

2. Real-Time Task Scheduling

- **Preemptive, Priority-Based Scheduling**: Critical tasks like **flight control** must have higher priority to ensure timely response to external factors (e.g., collision avoidance, dynamic weather conditions). Tasks that are less critical, such as **data logging** and **ground communication**, can be scheduled with lower priority, avoiding performance bottlenecks.
- **Distributed Task Allocation for Fleet**: The RTOS must manage tasks across multiple drones. **Centralized task distribution** allocates mission-critical tasks, such as surveillance or emergency response, to specific drones, while ensuring no drone’s task conflicts with others.
- **Coordinated Task Execution**: Ensures that drones in the fleet **maintain formation** and **synchronize actions** (e.g., coordinated surveillance, delivery), crucial for fleet operations.

3. Memory Management

- **Fixed Memory Partitions**: This ensures there are no unpredictable delays during memory allocation, which is vital in real-time systems where response time is crucial.
- **Memory Protection**: Prevents failure in one drone's tasks from affecting others in the fleet, ensuring stable operations.

4. Inter-Process Communication (IPC)

- **Message Queues & Semaphores**: IPC mechanisms ensure **safe data exchange** between tasks. For example, drones may share sensor data (e.g., GPS, LiDAR) for enhanced decision-making. IPC ensures data consistency and prevents race conditions in concurrent tasks.
- **Low-Latency Communication**: To manage real-time coordination between drones and ground stations, the RTOS must support **low-latency communication** (e.g., MQTT, DDS protocols) to allow rapid data exchange without delays.

5. Sensor Fusion and Data Processing

- **Real-Time Data Fusion**: Drones rely on multiple sensors (e.g., GPS, IMU, LiDAR) for navigation and situational awareness. The RTOS must **fuse data** from all drones in real time, providing a comprehensive environmental map.
- **Onboard Data Processing**: Offloading data processing to edge devices (onboard computers) ensures **low-latency decision-making** and allows drones to operate efficiently even in resource-constrained environments.

6. Safety and Redundancy Mechanisms

- **Failsafe Mode**: Ensures that if a drone faces a failure (e.g., sensor malfunction, communication issue), it can either be safely grounded or replaced by another drone in the fleet.
- **Collision Avoidance**: Drones must be able to detect and avoid collisions in real time. The RTOS should allow drones to share positional data and calculate optimal flight paths to avoid crashes.
- **Emergency Landing Protocols**: The RTOS ensures drones are capable of performing **safe emergency landings** in case of critical failure, reducing the risk to both the drone and its environment.

7. Power Management

- **Dynamic CPU Scaling**: To optimize performance while conserving battery life, the RTOS should adjust CPU power dynamically based on task demand.
- **Low-Power Mode for Non-Critical Tasks**: Non-essential tasks (e.g., data logging) should enter low-power modes when not needed, optimizing battery life for extended missions.

8. Communication and Coordination

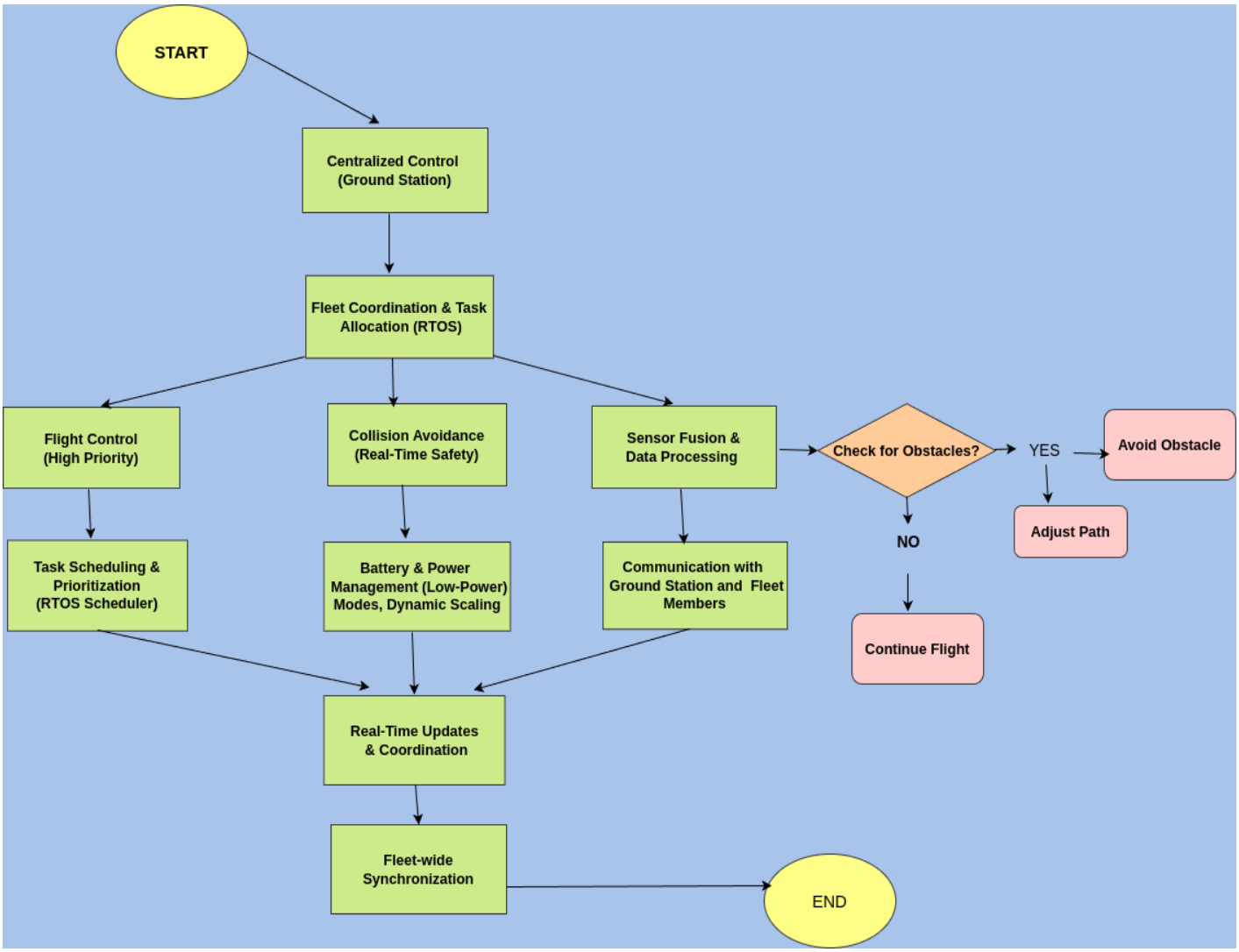
- **Fleet-Wide Coordination**: Drones must work in harmony to complete complex tasks like surveillance or large-area mapping. RTOS ensures that drones are aware of each other's positions and synchronize actions (e.g., **swarm intelligence**).
- **Centralized Control and Monitoring**: Ground stations or central controllers play a pivotal role in overseeing the entire fleet, relaying mission updates, and receiving real-time feedback from drones.

9. Fault Tolerance and Redundancy

- **Watchdog Timers**: These timers monitor critical functions (e.g., flight control) and can trigger a system reset if any function hangs or fails, ensuring continuous operation.

- **Redundancy in Systems**: Backup systems (e.g., multiple GPS modules, sensors) help ensure that the failure of one component does not compromise the entire operation.

Workflow Diagram for RTOS in Drone Fleet Management:



Workflow Explanation:

Centralized Control (Ground Station):
The ground station manages the fleet, assigns tasks, and monitors system health, providing mission updates and receiving feedback.

Fleet Coordination & Task Allocation (RTOS):
The RTOS prioritizes critical tasks (e.g., flight control, collision avoidance) while distributing less important tasks based on available resources.

Flight Control (High Priority):
The RTOS ensures safe navigation by controlling flight dynamics, reacting to environmental factors like obstacles and weather.

Collision Avoidance (Real-Time Safety):
Drones use sensors to monitor their surroundings, sharing positional data to prevent collisions in real time.

Sensor Fusion & Data Processing:
The RTOS fuses data from multiple sensors for accurate mapping and quick decision-making onboard, checking for obstacles,reducing reliance on the ground station.

Battery & Power Management:
The RTOS optimizes power usage, adjusting for task demands and placing non-essential tasks in low-power modes to extend battery life.

Communication with Ground Station and Fleet:
The RTOS ensures low-latency communication between drones and the central station, facilitating coordination and task synchronization.

Real-Time Updates & Coordination:
Drones share status updates, including battery levels and task progress, ensuring fleet awareness and smooth operation.

Fleet-wide Synchronization:
The RTOS synchronizes actions across drones, enabling coordinated operations like surveillance and mapping.

Challenges:

1. **Limited Hardware Resources:**
   Drones often have constrained processing power, memory, and battery life. Designing an RTOS that can perform real-time operations while conserving resources is a key challenge.
2. **Real-Time Task Scheduling:**
   Ensuring that high-priority tasks like flight control and collision avoidance are handled

without delay, while efficiently managing less critical tasks, can be difficult in a multi-drone environment.

3. **Inter-Drone Communication:**
   Coordinating real-time communication and data sharing between drones in a fleet without causing network congestion or delays is a significant challenge.
4. **Fault Tolerance & Redundancy:**
   Ensuring continuous operation in case of system failures (e.g., sensor malfunctions or communication breakdowns) is critical for safety and reliability.
5. **Sensor Fusion & Data Processing:**
   Efficiently processing data from multiple sensors in real time to ensure accurate environmental mapping while minimizing latency is challenging, especially with limited computational resources.
6. **Power Management:**
   Balancing power consumption and task execution, particularly when drones need to operate for extended periods without frequent recharging, is essential for long-duration missions.
7. **Scalability:**
   Managing and synchronizing tasks across a growing fleet of drones while maintaining performance and avoiding conflicts in task execution becomes more complex as the fleet expands.

Solutions:

1. **Optimized Resource Management:**
   Implementing a lightweight microkernel architecture ensures minimal overhead, allowing the RTOS to handle essential tasks efficiently without consuming too much processing power or memory. Additionally, dynamic CPU scaling and low-power modes help conserve battery life during non-critical operations.
2. **Preemptive Task Scheduling:**
   A priority-based, preemptive scheduling mechanism ensures that critical tasks such as flight control and collision avoidance are processed with high priority, while less important tasks like data logging can be scheduled based on available resources, preventing performance bottlenecks.
3. **Low-Latency Communication Protocols:**
   Using efficient communication protocols like MQTT or DDS, along with message queues and semaphores for inter-process communication, ensures low-latency data exchange between drones and the ground station. This allows for smooth coordination and real-time updates.

4. **Redundancy & Failsafe Mechanisms:**
   Implementing redundancy in critical systems (e.g., multiple sensors and backup communication links) ensures that failure in one component doesn't compromise the entire system. Failsafe modes and watchdog timers trigger automatic recovery or safe shutdowns in case of critical failures.
5. **Efficient Data Fusion Algorithms:**
   Advanced algorithms for sensor fusion (e.g., Kalman filters) allow for real-time data processing, integrating inputs from GPS, IMU, and LiDAR sensors. Offloading computation to onboard processors ensures that decisions are made quickly with minimal delay.
6. **Energy-Efficient Power Management:**
   Implementing dynamic power scaling for high-demand tasks and low-power modes for non-essential tasks ensures optimal energy usage, extending mission durations while maintaining performance for critical operations.
7. **Scalable Fleet Coordination:**
   A distributed task management system allows for efficient allocation of tasks to drones in a large fleet, ensuring synchronization and preventing conflicts. Task coordination algorithms optimize resource allocation and communication, ensuring smooth fleet-wide operations.

Conclusion:

Designing a lightweight real-time operating system for a drone fleet managing autonomous flight involves addressing several key challenges, such as limited resources, real-time scheduling, communication, and power management. By implementing solutions like a microkernel architecture, efficient task scheduling, low-latency communication, redundancy mechanisms, and optimized power management, these challenges can be mitigated. The resulting RTOS would ensure that drones can operate safely, efficiently, and reliably, enabling them to perform complex, coordinated missions even in dynamic and resource-constrained environments.

Scenario 2: Embedded Operating System for a Smart Home Device

Overview:

The embedded operating system for a smart home device needs to manage various tasks such as sensor data processing, device control, network communication, and user interactions. It must be lightweight and reliable, as the smart home device often operates on limited hardware and requires continuous operation for safety and convenience.

Key Features and Architecture for Embedded OS in Smart Home Device:

1.**Microkernel Architecture:** The OS utilizes a microkernel approach, where only essential functionalities such as task scheduling, device management, and communication are Implemented. This reduces system complexity and resource consumption, crucial for embedded systems.

2.Real-Time Task Scheduling: The OS employs a real-time scheduling mechanism to ensure time-sensitive tasks, like responding to security alerts or activating devices (lights, locks), are prioritized. This guarantees a prompt response to environmental changes.

3.Low-Power Management: To optimize battery life and power usage, the system dynamically adjusts the power states based on activity. Non-critical tasks enter low-power modes, allowing the system to be energy-efficient while remaining operational.

4.Sensor Integration and Data Processing: Multiple sensors (motion detectors, temperature, humidity) are integrated, with real-time data processing locally. The system can react quickly to changes, triggering actions like adjusting the thermostat or activating security systems.

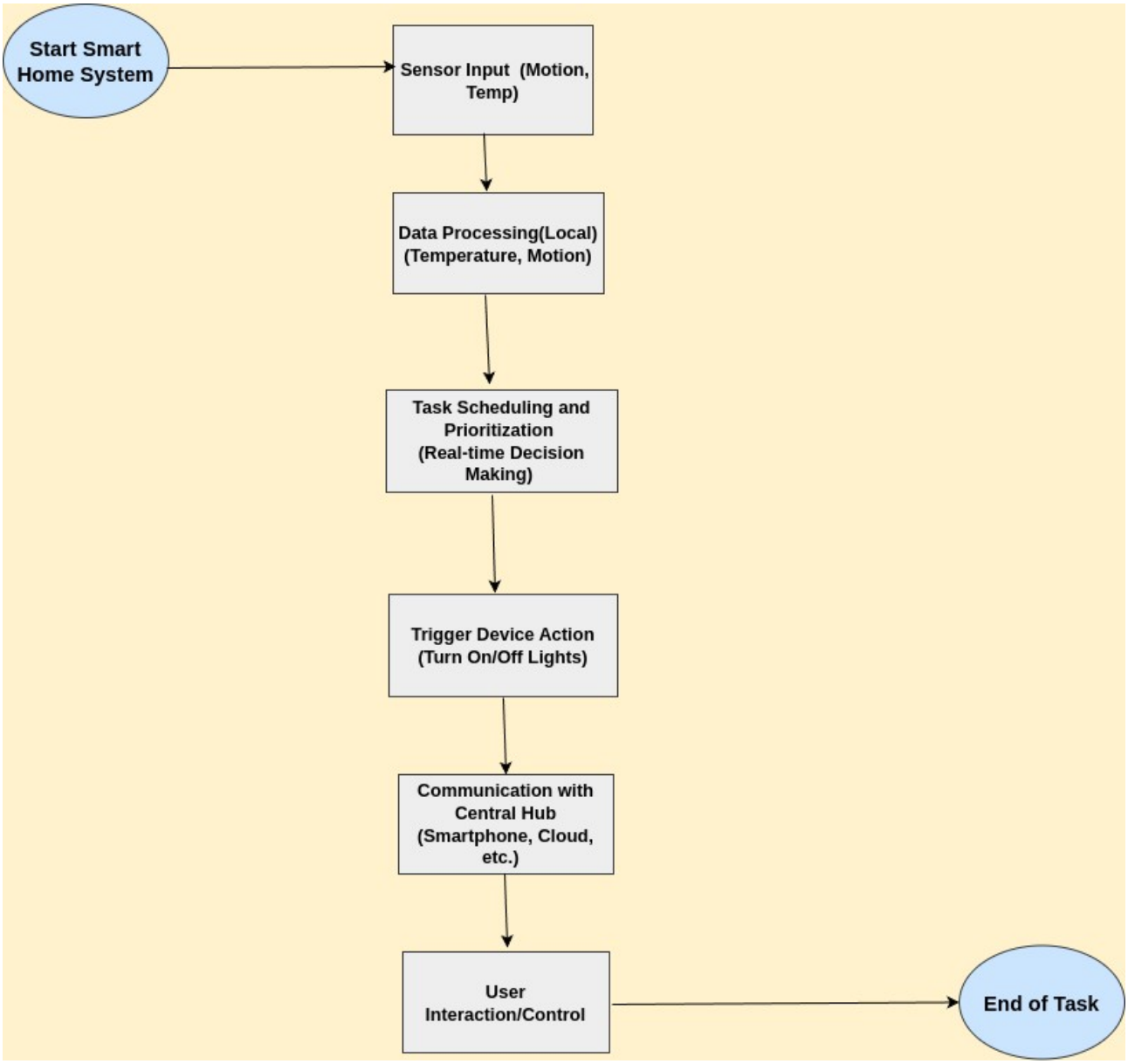
5.Inter-Process Communication (IPC): The OS facilitates efficient communication between processes through IPC mechanisms like message queues or semaphores, ensuring that tasks such as sensor monitoring and device control can operate concurrently without delays.

6.Wireless Communication Support: The system supports communication protocols like MQTT or HTTP, enabling it to seamlessly interact with other smart devices, control hubs, and cloud services for centralized management and remote control.

7.User Interface and Control: A simple user interface (through mobile apps or voice assistants) is supported, enabling users to control devices, monitor environmental data (temperature, security status), and receive updates on system performance.

8.Real-Time Response: The OS is designed to handle real-time processing of critical data, such as responding to security breaches or adjusting home devices (e.g., lights, locks, heating) based on immediate input or environmental changes.

Workflow Diagram of Embedded Operating System for a Smart Home Device:



Workflow Explanation:

Sensor Input gathers real-time environmental data (e.g., motion, temperature, humidity).

- **Data Processing** occurs locally, where the sensor data is processed and used for decision-making (e.g., deciding to turn on a light based on motion detection).

- **Task Scheduling** ensures that real-time tasks (like security alerts or user input) are given priority over non-critical tasks (like firmware updates).
- **Device Action** triggers changes in the smart home system (e.g., lights, thermostat).
- **Communication** with the central hub (smartphone app or cloud service) provides updates or allows for remote control of the device.
- **User Interaction** allows users to interact with the system, either through manual control or voice commands.

This workflow ensures that the system operates smoothly, responding to user inputs and environmental changes in real-time while managing tasks and conserving energy efficiently.

Challenges:

1. **Limited Resources:**
   The embedded system often operates with limited CPU power, memory, and storage, making it difficult to implement complex operating systems.
2. **Real-time Responsiveness:**
   Ensuring real-time processing of sensor data and user commands is critical in a smart home system. Delays could lead to undesirable outcomes, like missed security alerts or delayed lighting control.
3. **Power Management:**
   The device needs to operate continuously without frequent recharging. Efficient power management is required to ensure long-term functionality on battery-powered devices.
4. **Device Interoperability:**
   The smart home device may need to interact with various devices from different manufacturers, requiring robust communication protocols and standardized interfaces.

Solutions:

1. **Lightweight Kernel:**
   The microkernel architecture ensures minimal overhead and only the essential functionalities (e.g., task scheduling, sensor data handling, network communication) are present. This helps in overcoming limited system resources.
2. **Real-Time Scheduling:**
   The RTOS implements preemptive task scheduling with high-priority tasks (such as security

alerts or motion detection) and low-priority tasks (like updating firmware) to ensure timely and reliable responses.

3. **Efficient Power Management:**
   Power management techniques, such as dynamic voltage scaling and low-power idle modes for non-essential tasks, ensure that the device can run efficiently for extended periods.
4. **Standardized Communication Protocols:**
   Support for communication protocols like MQTT and HTTP ensures seamless interaction with other devices in the smart home ecosystem, overcoming interoperability challenges.

Conclusion:

An embedded operating system for a smart home device must be lightweight, real-time, and energy-efficient. By utilizing a microkernel architecture, efficient power management, real-time task scheduling, and standardized communication protocols, the system can effectively handle device control, sensor integration, and network communication. The design ensures the smart home device operates reliably, with minimal latency, while conserving power and remaining responsive to user and environmental inputs.

Scenario 3: A Mini OS for a Gaming Console with Limited Hardware Resources

Overview:

A mini OS for a gaming console must prioritize fast performance, efficient memory usage, and smooth gameplay execution on limited hardware. The OS should optimize CPU and GPU usage, handle game rendering, and ensure seamless controller inputs for an immersive gaming experience.

Key Features and Architecture for Mini OS in Gaming Console:

Hybrid Kernel:

- Combines monolithic and microkernel elements for performance and modularity.
- Game-critical processes (e.g., rendering, input handling) run directly in kernel mode for faster execution.

Task Scheduling for Gaming Performance:

- Real-time scheduling for graphics rendering and physics calculations.
- Round-robin scheduling for secondary tasks like audio processing and network communication.

- **Memory Management for Optimized Performance:**

- Dynamic memory allocation for game assets.
- Cache management to speed up loading times.

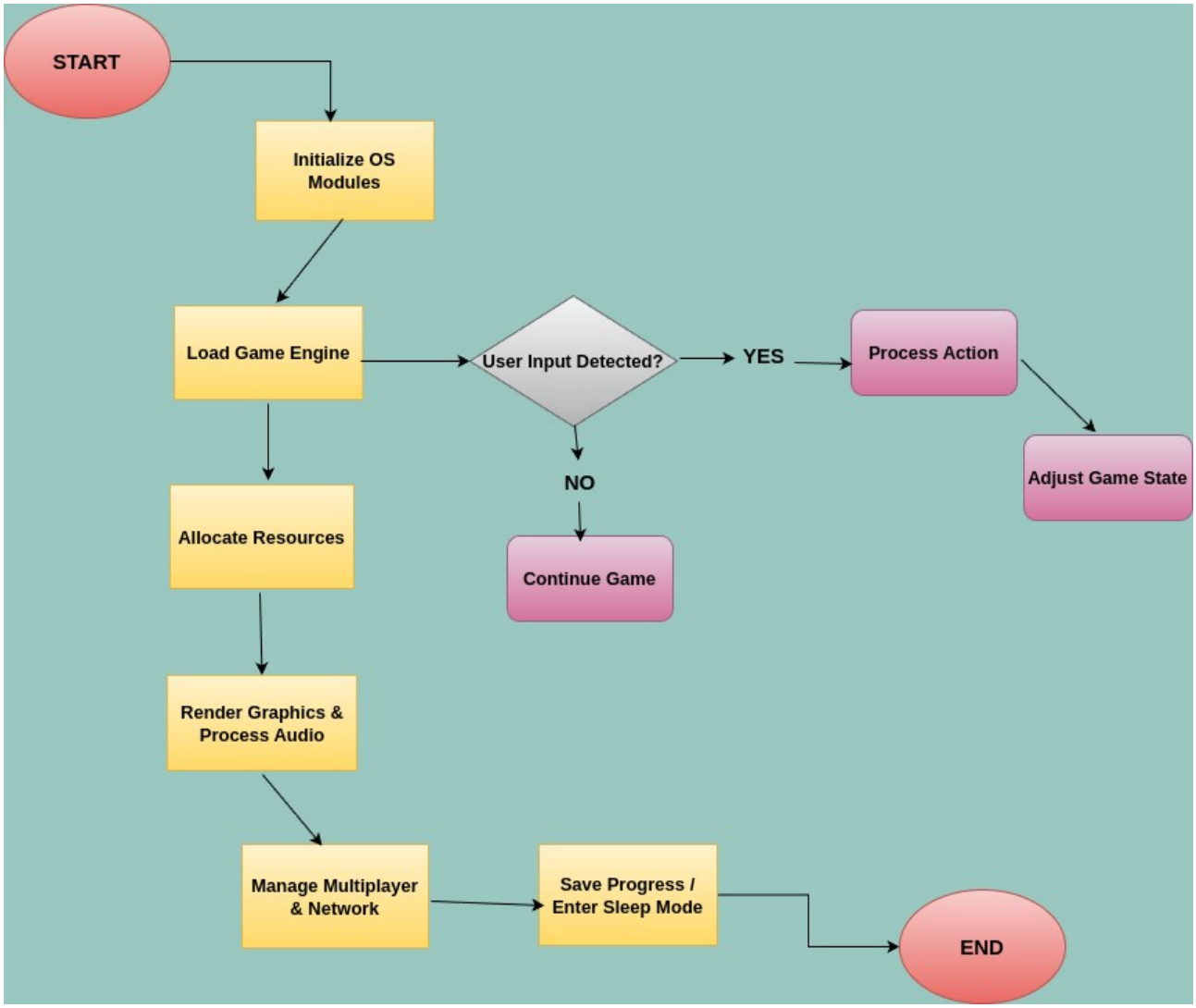
Graphics & Input Handling:

- Direct GPU access for high-speed rendering.
- Low-latency controller input processing for real-time gameplay.

Networking & Multiplayer Support:

- Implements low-latency networking protocols for online gaming.
- Uses predictive input buffering to reduce lag.

Workflow Diagram for Mini OS in Gaming Console with Limited Hardware Resources:



Workflow Explanation:

1. **Initialize OS Modules**: The system loads essential operating system components, including memory management, task scheduling, and device drivers.
2. **Load Game Engine**: The game engine and necessary resources (graphics, sounds, physics engine) are loaded into memory.
3. **User Input Detected?**: The system continuously checks for user input (controller, keyboard, or motion sensors).
    - **Yes → Process Action**: If input is detected, the system processes the action and updates the game state.
    - **No → Continue Game**: If no input is detected, the game continues running as per its logic.
4. **Allocate Resources**: The OS dynamically assigns CPU, GPU, and memory resources to different processes, prioritizing real-time rendering and physics calculations.
5. **Adjust Game State**: If an action requires changes (e.g., character movement, weapon firing), the game engine updates the state accordingly.
6. **Render Graphics & Process Audio**: The system renders the updated game visuals and processes game sounds in real time.
7. **Manage Multiplayer & Network**: If the game includes online multiplayer, network communication is handled efficiently with predictive buffering to minimize lag.
8. **Save Progress / Enter Sleep Mode**: The system periodically saves progress and can transition into a low-power state when inactive.
9. **End**: The game session ends when the user exits or the system powers down.

Challenges:

1. **Limited Hardware Resources**: Gaming consoles have fixed CPU, RAM, and storage, requiring efficient optimization.
2. **Low Latency Requirement**: Games need real-time responsiveness for smooth gameplay and minimal input lag.
3. **Graphics Processing Constraints**: Rendering high-quality visuals within hardware limitations is challenging.
4. **Memory Management**: Allocating memory efficiently between game assets, system processes, and caching.
5. **Networking Latency**: Online multiplayer requires fast data synchronization with minimal lag.

Solutions:

1. **Hybrid Kernel Approach**: Combining monolithic and microkernel elements ensures performance and modularity.
2. **Real-Time Scheduling**: Prioritizing rendering and input processing to maintain smooth frame rates.
3. **Optimized Memory Management**: Using caching, memory pooling, and compression to maximize performance.
4. **Efficient Graphics Handling**: Direct GPU access and shader optimizations to enhance rendering speed.
5. **Network Optimization**: Implementing predictive buffering and low-latency protocols to reduce lag in multiplayer gaming.

Conclusion:

A mini OS for a gaming console with limited hardware resources must be highly optimized for performance, efficiency, and responsiveness. Using a hybrid kernel, the system ensures real-time task execution while maintaining modularity. Efficient task scheduling, memory management, and graphics processing enable smooth gameplay, while low-latency networking enhances the multiplayer experience. By addressing hardware limitations with smart optimizations, the OS ensures a seamless gaming experience within resource constraints.

Answer to Question No. 2

Extracting the Reference String:

My **student ID** is **0222220005101039**. Using all the digits as a reference string, we get:

📌 **Reference String:** `0 2 2 2 2 2 0 0 0 5 1 0 1 0 3 9`
📌 **Number of Frames:** `3`

Page Replacement Algorithms:

1. FIFO (First-In-First-Out) Algorithm

- FIFO replaces the oldest page in memory when a new page needs space**.**

Simulation Table

Step	Page	Frames	Hit/Miss
1	0	0	Miss
2	2	0,2	Miss
3	2	0,2	Hit
4	2	0,2	Hit
5	2	0,2	Hit
6	2	0,2	Hit
7	0	0,2	Hit
8	0	0,2	Hit
9	0	0,2	Hit
10	5	2,0,5	Miss
11	1	0,5,1	Miss (2 removed)
12	0	5,1,0	Miss (0 removed)
13	1	5,0,1	Hit
14	0	5,0,1	Hit
15	3	0,1,3	Miss (5 removed)
16	9	1,3,9	Miss (0 removed)

📌 **FIFO Performance:**
Total Hits: 9
Total Misses: 7

Hit Ratio: 9/16 = 0.5625 (56.25%

Miss Ratio: 7/16 = 0.4375 (43.75%)

2. Optimal Page Replacement Algorithm

- Optimal replaces the page that will not be used for the longest period in the future.

Simulation Table

Step	Page	Frames	Hit/Miss
1	0	0	Miss
2	2	0,2	Miss
3	2	0,2	Hit
4	2	0,2	Hit
5	2	0,2	Hit
6	2	0,2	Hit
7	0	0,2	Hit
8	0	0,2	Hit
9	0	0,2	Hit
10	5	0,2,5	Miss
11	1	0,5,1	Miss (2 removed)
12	0	5,1,0	Hit
13	1	5,1,0	Hit
14	0	5,1,0	Hit
15	3	1,0,3	Miss (5 removed)
16	9	0,3,9	Miss (1removed)

📌 **Optimal Performance:**

Total Hits: `10`
Total Misses: `6`
Hit Ratio: 10/16 = 0.625 = 62.5%
Miss Ratio: 6/16 =0.375 = 37.5 %

3. LRU (Least Recently Used) Algorithm

- LRU replaces the least recently used page in memory when a new page needs space.

Simulation Table

Step	Page	Frames	Hit/Miss
1	0	0	Miss
2	2	0,2	Miss
3	2	0,2	Hit
4	2	0,2	Hit
5	2	0,2	Hit
6	2	0,2	Hit
7	0	0,2	Hit
8	0	0,2	Hit
9	0	0,2	Hit
10	5	0,2,5	Miss
11	1	2,5,1	Miss (0 removed)
12	0	5,1,0	Miss (2 removed)
13	1	5,0,1	Hit
14	0	5,0,1	Hit
15	3	0,1,3	Miss (5 removed)
16	9	1,3,9	Miss (1 removed)

📌 **LRU Performance:**

Total Hits: 9
Total Misses: 7
Hit Ratio: 9/16 = 0.5625 (56.25%

Miss Ratio: 7/16 = 0.4375 (43.75%)

Final Comparison

Algorithm	Hits	Misses	Hit Ratio	Miss Ratio
FIFO	9	7	56.25%	43.75%
OPTIMAL	10	6	62.5%	37.5%
LRU	9	7	56.25%	43.75%