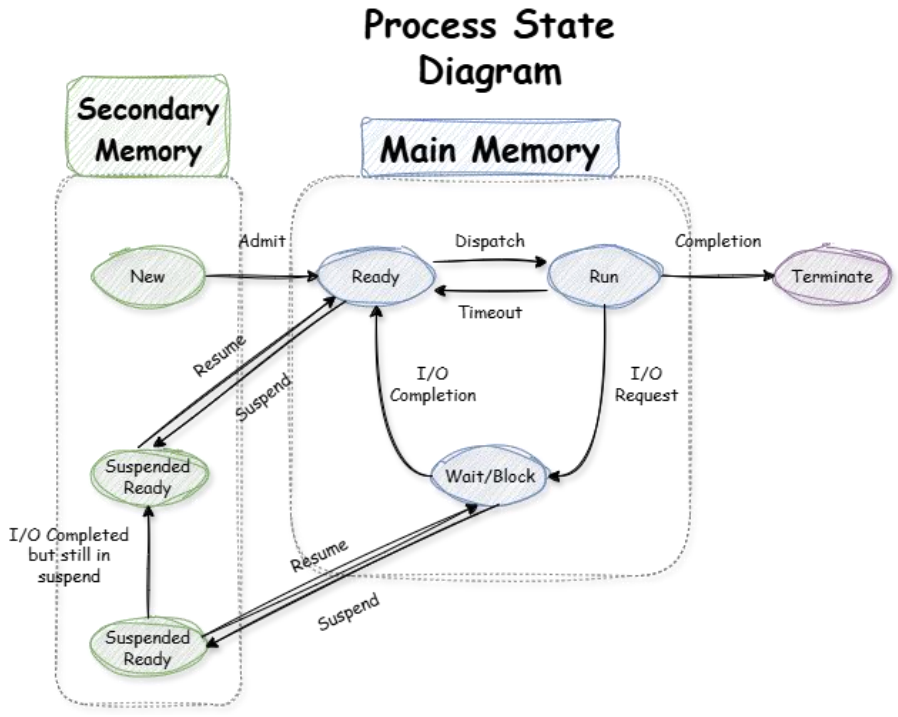
Layer 1: Classic OS Problem Analysis

Part A: Process Management Fundamentals

- Create a comparison table: Processes vs Threads.

	Processes	Threads
Definition	Execution of any program is a process, usually independent.	Exists as a subset of a process.
Memory	Separate address space when running.	Shared address space within a process.
Communication	Communicate through IPC; it takes more time to communicate between processes.	Communicate freely by modifying shared variables, which takes less time for the communications.
Size	Heavyweight; processes consume more memory.	Lightweight; each thread in a process shares code, data, and resources.
Context Switching	Takes more time for context switching.	Takes less time for context switching.

- Draw and explain the process state diagram (refer to Figure 1).



- Explain two IPC methods with real-world examples.

- **Shared memory** is a method where two or more processes access a common memory space, allowing direct reading and writing for communication. It is considered the fastest form of inter-process communication (IPC) since data exchange happens instantly without message passing. A real-world example can be seen in video games, where one process handles graphic rendering while another manages the game's physics and logic. After a game action, the logic process writes new object coordinates to the shared memory, and the rendering process immediately reads this data to update the screen, ensuring smooth and efficient performance.

- **Message queues** allow multiple processes to exchange data by sending and receiving structured messages in a first-in-first-out (FIFO) manner. Using system calls, processes can send messages to a queue, enabling asynchronous communication where the sender and receiver do not need to be active simultaneously. A real-world example is an online food ordering system, where a message queue acts like a central order board in a kitchen. When a customer places an order, the web server posts it to the queue instead of waiting for it to complete, allowing backend services like inventory, payment, and email to independently process the order. This approach ensures asynchronous operation, prevents order loss, and keeps the system running smoothly even if one service temporarily fails.

Part B: Memory Management Basics

- Explain the differences between paging and segmentation.
  - Paging and segmentation are both memory management techniques, but differ in how they organize memory. Paging divides physical memory into fixed-size frames and logical memory into equal-sized pages, simplifying management and removing external fragmentation, though it may cause internal fragmentation when the last page isn't fully used. The OS and Memory Management Unit (MMU) handle this automatically. Segmentation, on the other hand, divides logical memory into variable-sized segments based on program structure (code, data, stack), avoiding internal fragmentation but introducing external fragmentation due to uneven segment sizes. It is visible to the programmer and supports better protection and sharing. Modern systems often combine both, using segmentation for logical organization and paging for efficient physical memory use.

- Describe the TLB and its impact on performance.
  - The Translation Lookaside Buffer (TLB) is a small, high-speed cache that stores recent virtual-to-physical address translations to speed up memory access in a paged virtual memory system. Without it, each memory access would require multiple slow lookups in the page table stored in main memory. The TLB improves performance by providing

quick access to these translations, often in just one clock cycle during a TLB hit, which greatly reduces effective memory access time. Conversely, a TLB miss forces the system to access the page table, update the TLB, and then perform the memory access, taking tens to hundreds of clock cycles. Thus, even a small TLB with a high hit rate can significantly enhance overall system performance.

- Calculate Effective Memory Access Time (EMAT) for the scenario in Figure 2.
  Given T_{mem} = 100 ns, T_{TLB} = 10 ns, hit rate = 0.8, miss rate = 0.2:
  EMAT = (0.8 × (10 + 100)) + (0.2 × (10 + 100 + 100))
  EMAT = 0.8·110 + 0.2·210 = 130 ns

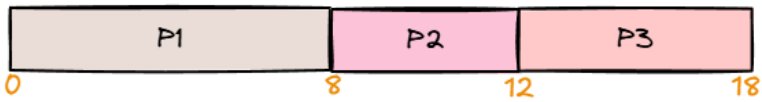
Part C: CPU Scheduling Principles

- Compare FCFS vs. Round Robin scheduling (see Figure 3).

FCFS	Round Robin
Non-preemptive scheduling method.	Preemptive scheduling method.
Processes are executed in the order they arrive (first come, first served).	Each process gets a fixed time slice (quantum) before moving to the next.
It can be unfair because long jobs delay shorter ones.	Fair because each process gets equal CPU time.
Fewer context switches.	More context switches due to frequent time slicing.
Poor performance due to high average wait times.	Better performance and shorter average waiting time for time-shared systems, but performance depends on the chosen time quantum.

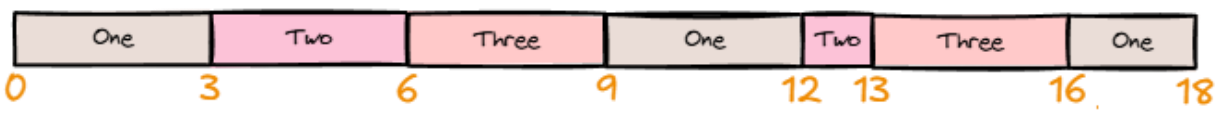
- For P1(8ms), P2(4ms), P3(6ms), create Gantt charts and find avg. waiting time for FCFS and Round Robin (quantum=3ms).

FCFS: P1 = 8 ms, P2 = 4 ms, P3 = 6 ms



Average waiting time = ((8 - 8) + (12 – 4) + (18 - 6)) / 3 = **6.67ms**

Round Robin: P1 = 8 ms, P2 = 4 ms, P3 = 6 ms, quantum = 3 ms



Average waiting time = ((18 - 8) + (13 - 4) + (16 - 6)) / 3 = **9.67ms**

Part D: Classic Synchronization Problems

- Describe the Dining Philosophers problem (see Figure 4) and its solutions.
  - The Dining Philosophers Problem demonstrates the challenges of synchronization and resource sharing among concurrent processes. Five philosophers sit around a table with one fork between each pair, and since each needs two forks to eat, poor coordination can lead to deadlock or starvation. To address this, three of the best solutions are the **Monitor-Based**, **Semaphore-Based**, and **Arbitrator (Waiter)** approaches. The Monitor-Based solution uses monitors to manage access to forks, allowing a philosopher to eat only when both forks are available, effectively preventing deadlock and starvation. The Semaphore-Based solution applies binary semaphores for forks and a counting semaphore to limit the number of philosophers eating simultaneously, ensuring efficient low-level control and avoiding circular waiting. Meanwhile, the Arbitrator (Waiter) solution introduces a central authority that grants permission before philosophers pick up forks, guaranteeing fairness and preventing deadlock at the cost of slight overhead.

- Explain the Producer-Consumer problem (see Figure 5) and the use of semaphores.
  - In this synchronization problem, the producer generates data items and places them in a bounded buffer, while the consumer retrieves them for processing. The issue arises when the producer tries to add to a full buffer or the consumer tries to remove from an empty one. Semaphores (empty, full, and mutex) are used to coordinate access. Mutex ensures mutual exclusion, while empty and full track buffer availability, preventing race conditions and maintaining proper synchronization between producer and consumer.

- Analyze Reader-Writer problem scenarios.
  - This synchronization problem involves managing access to shared data in an online ticket booking system for concerts or movies. The database stores information about available seats, and there are two types of users: viewers and bookers. Viewers can browse available seats at the same time without causing issues since they are only reading data. However, when a booker reserves a seat, no other viewer or booker should access that same seat information during the update to prevent inconsistencies. If synchronization is not properly implemented, two users might end up booking the same seat simultaneously, or a viewer might see outdated availability. This leads to race conditions and data inconsistency, and if access handling is unfair, it may also cause starvation, where either viewers or bookers are forced to wait indefinitely.