Comprehensive Report

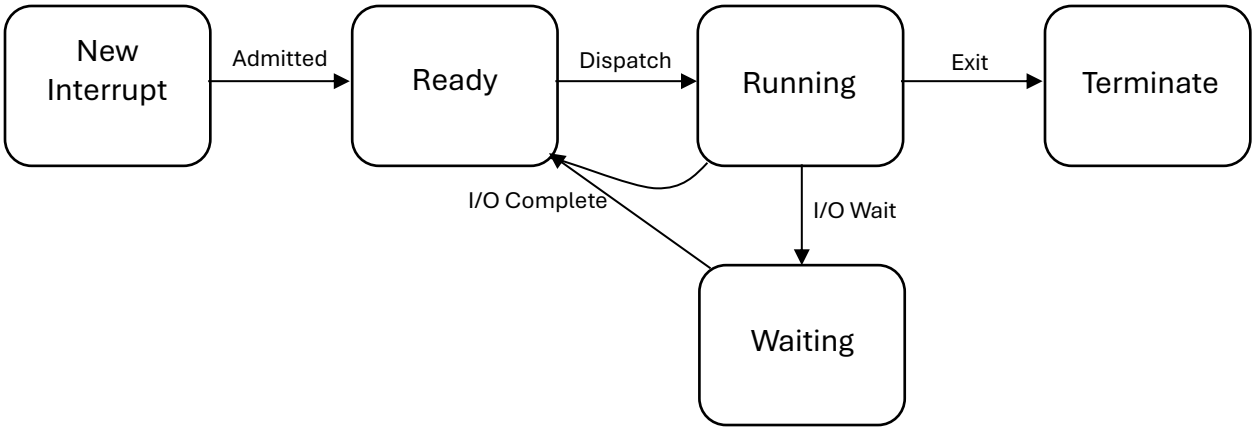
Layer 1: Classic Problems Analysis

Part A: Process Management Fundamentals

- **Process vs Thread Comparison Table**

Process	Thread
A unit of execution of a program.	Unit of execution within a process. A process can have any number of threads, from just one thread to many threads.
Process does not share resources with other process that belong to the same program.	Threads share resources with other threads that belong to the same process.
Communication between processes that belong to the same program is slow.	Communication between the threads that belong to the same process is fast.
Every process has its own separate memory space.	Threads within the same process share the same memory space.
An error in one process does not affect others.	One thread error can crash the entire process
Example: A web browser application.	Example: Multiple tabs within a web browser.

- **Process State Diagram**



For a program to be executed, a process is created for that program. The **New state** corresponds to a process that has just been defined. At this point, the OS has performed the necessary actions to create the process but has not committed itself to the execution of the process. The OS will then move a process from the New state to the **Ready state** when the process is prepared to execute when given the opportunity. When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher. The process will now proceed to the **Running state** and is currently

being executed. From the Running state, a process can go to three different states: Termination state, Waiting state, or back to the Ready state. If the process indicates that it has completed or if it aborts, the OS will move the process to the **Termination state**. The process will now be released from the pool of executable processes by the OS. A process is put in the **Waiting state** if it requests for something for which it must wait, such as the completion of an I/O operation or waiting for another process to provide data. The process will then move to the Ready state when the data requested is now given or the event for which it has been waiting occurs. For a process to go back to the Ready state from Running state, the most common reason would be that the running process has reached the maximum allowable time for uninterrupted execution. Another cause would be the different levels of priority of different processes. A currently running process that has a low priority can be swap by a high priority process in the Ready state.

- **Inter-process Communication**

Inter-process communication via a **shared memory** is memory-based communication. In this method processes access a common memory area to exchange data and information. A real-world example using the shared memory method are web browsers. Web browsers like Chrome run each tab in its own different process to prevent one from crashing the entire application. These separate processes use shared memory to exchange data quickly, such as when a process need to acquire information from another different tab.

Another method for IPC is **message queue**, which allow processes to send and receive messages in a queue like manner. The sending and receiving processes are independent of each other and do not need to be running at the same time. Often managed by the operating system kernel, that acts as a buffer for asynchronous message passing. Amazon, Shopee, and other e-commerce platforms are prime real-world examples of systems that heavily use message queues. This approach ensures that a high volume of orders can be accepted and processed smoothly.

Part B: Memory Management Basics

- **Difference between Paging and Segmentation**

With paging, each process is divided into relatively small, fixed-size pages. The logical address space is a single, one-dimensional space. Segmentation provides for the use of variable-sized segments. The logical address space is a collection of segments, representing a two-dimensional space.

- **Translation Lookaside Buffer (TLB)**

A hardware cache within a computer's Memory Management Unit (MMU) that stores recently used mappings of virtual memory addresses to physical memory addresses. By caching frequently used address translations, the TLB provides a quicker way to find physical addresses, significantly reducing memory access latency and improves the overall system performance.

- **Calculate Effective Memory Access Time (EMAT)**

Based on the figure 2, here is how we solve the EMAT.

Given: T_{mem} = 100 ns

T_{TLB} = 10 ns

Hit Rate = 80%

EMAT = T_{TLB} + ((HitRate * T_{mem})+ (1-HitRate) * 2T_{mem})

= 10 + ((0.80 * 100) + (0.20) * 2(100))

= **130 ns**

Part C: CPU Scheduling Principles

- **First Come First Serve (FCFS) vs Round Robin (RR) Scheduling Comparison**

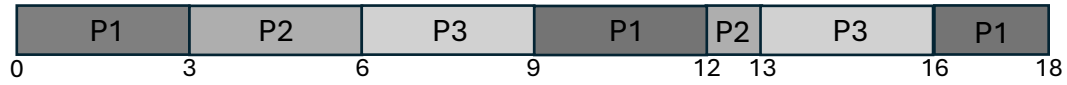
FCFS	RR
The processes get assigned to the CPU according to their order of arrival.	Each process gets a fixed time slice, then the next process runs.
Non-preemptive scheduling method.	A preemptive scheduling method.
Suitable for batch systems.	Suitable for interactive systems.
Only causes few context-switches.	Causes many context-switches.

- **Gantt Charts**

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

average waiting time = (0 + 8 + 12)/3 = **6.67 ms**

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



P1 waiting time = 0 + 6 + 4 = 10

P2 waiting time = 3 + 6 = 9

P3 waiting time = 6 + 4 = 10

average waiting time = (10 + 9 + 10)/3 = **9.67 ms**

Part D: Classic Synchronization Problems

- **Dining Philosophers Problem**

There are five philosophers sitting at a roundtable. Each philosophers have their own bowl of rice and a single chopstick between each pair of philosophers. They only have two states: eating and thinking. To eat, the philosopher needs to pick up both chopsticks – one on the left and one on the right. One cannot pick up a chopstick that is already in the hand of another philosopher. After eating, the philosopher will put down both chopsticks and will switch to the thinking state. A problem occurs when every philosopher simultaneously picks up their left chopstick, no one can get the second chopstick causing an endless loop of

waiting with each other leading to deadlock. Starvation also occurs when a philosopher waits too long to be able to eat. This is a classic experiment about resource allocation and concurrency, illustrating the challenge of preventing deadlock and starvation.

Solutions:

Using A Monitor	Asymmetry	Resource Hierarchy
A philosopher must ask the monitor for permission to pick up chopsticks. The monitor can prevent a philosopher from picking up chopsticks if a neighbor is already eating, preventing deadlock.	Odd numbered philosophers will pick up their left chopstick first, even numbered pick up their right first. This will break the endless waiting loop.	Each chopstick is assigned to a unique number – 1 to 5. Each philosopher must always pick up the lower-numbered chopstick first. Eliminates the circular waiting, preventing deadlock.

- **Producer-Consumer Problem**

The Producer- Consumer problem also called as Bounded-Buffer problem is one of the classic problems of synchronization. There is a shared buffer that has n available of slots and each slot can store one unit of data. The *producer* process creates data and place them in the shared buffer. The *consumer* process takes data from the buffer and "consume' it. Since the buffer has limited capacity (n), the *producer* must stop putting data in the buffer if it is full and wait until a slot is available. Consequently, the *consumer* must wait if the buffer has no data or empty. These two process must not access the buffer at the same time to prevent race condition that could lead to corrupting of data or inconsistent results. Using semaphore provide one of the best and efficient solutions to it. Semaphore is an integer variable that has two standard operations: wait () and signal (). The binary semaphore is used for mutual exclusion, ensuring only one process manipulates the buffer at a time. Counting semaphore "empty" has the initial value as the number of slots of the buffer, since initially all slots are empty. Counting semaphore "full" has an initial value of zero, it counts the filled slots. The producer must wait if empty is equal to 0, it means the buffer is full. The consumer must wait if full is equal to 0, it means the buffer is empty.

- **Reader-Writer Problem**

This is another classic synchronization problem that deals with managing access to shared data by multiple processes, ensuring data integrity while allowing maximum concurrency. Suppose we have a database or a file that is shared among several concurrent processes. These processes belong to two types: readers and writers. Readers can read the shared database at the same time, it is safe. In contrast, when a writer is writing, no other writer must write and no reader must read the shared data. A problem occurs when synchronization is not done properly, leading to race conditions. If a reader reads the data in the middle of update. The reader may see corrupted data or an old value, producing inconsistent results. Another problem would be starvation, either the reader or writer would starve if handling of access is not fair.