Layer 1

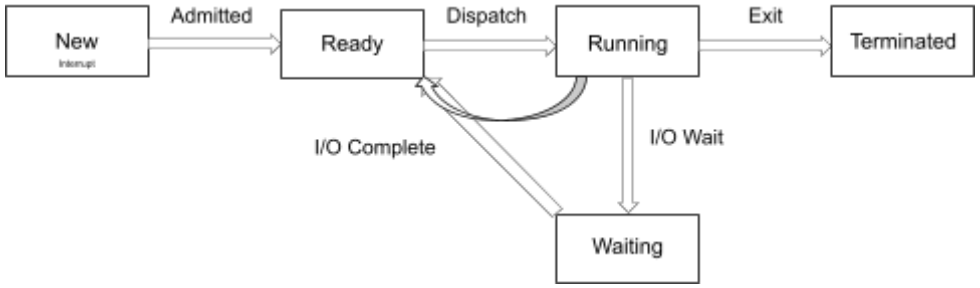
A. Process Management

Comparison Table : Processes vs Threads

	Process	Thread
Definition	Process means a program in execution	Thread means a part of a process
Termination TIme	It takes more time to terminate	Take less time to terminate
Creation Time	More time for creation because its of more setups like PCB and memory	Less time for creation because they share resources to each other
Switching	More time for switching due to changes in memory and state	Less time for switching as they share address space
Communication	Less efficient in terms of communication by using IPC mechanism	More efficient in terms of communication by sharing data in the same memory space
Multiple program	Its holds concepts of multi-process	It doesn't need multi programs for multiple threads because a single process consists of multiple threads
Memory Usage	Every process runs in its own memory	Threads share memory
Weight	It has more memory and control block making it heavyweight	Lightweight as thread in process shares code, data and resources
Switching Mechanism	It uses an interface in an operating system	It may not require to call in the operating system (user-level)
Blocking	Even if one process is block the other process will not be affected in execution	If a user-level thread is block, then all other user-level threads are also block

Control Blocks	Its owns process control block (PCB), Stack, and Address Space	It has a parents' PCB, thread control block, Stack, and common address space
System Call	System call in involved in it (Kernel Calls)	Doesn't involved in system call but it is created using Application Programming Interfaces (APIs)
Data Sharing	It doesn't share data with each other it needs IPC	Threads shares data with each other

Process State Diagram



New State - This is where the process is created or loads the program code and setting up for it to run, once it is complete it will be admitted and moved to the Ready State

Ready State - When the process is admitted to this state it means that it's ready to be executed but still must wait for the CPU to be available. The process will stay in the ready queue managed by the scheduler then selects one process from the queue dispatching it to the Running State after checking the CPU availability.

Running State - The process is currently using the CPU to execute its given instructions, this is where the process may perform one of the following procedures - calculate, read or write data, or make system calls. After finishing the execution it will exit towards the terminated state.

Waiting State - The completed process cannot be executed due to waiting for external inputs to happen or file reading, for example if the process may be halted waiting for a keyboard input, printer access, or data retrieval. After the I/O is done it will proceed to Ready state

Terminated - The process has finished execution and is removed from the scheduling pool.

Inter-Process Communication

- This is how different processes communicate or share data with each other in an operating system. There are two types of IPC methods and these are shared memory and message passing.

Shared memory

In a shared memory inter-process communication (IPC) model, two or more processes share a portion of the same physical memory space to read and write data directly, allowing faster communication by avoiding kernel involvement. A real-life example is a group of students editing a shared Google Docs file, where everyone can view and update content in real time. However, if multiple users edit the same section simultaneously, confusion may occur. Similarly, in shared memory systems, synchronization tools are used to coordinate access and prevent data corruption.

Massage passing

In message passing inter-process communication (IPC), processes exchange data by sending and receiving messages through the system kernel instead of directly accessing each other's memory space. While generally slower than shared memory, this method is safer since each process operates within its own memory area, avoiding accidental interference and eliminating the need for synchronization. Message passing is ideal for distributed systems like client-server architectures and email systems. A real-life example is a customer sending a product for repair—the product represents the message being sent and received, ensuring controlled and secure communication between both parties.

B. Memory management

In the Operating system there are two methods in memory management which are paging and segmentation, they both use for an efficient allocation and access to memory but they work differently from each other and do different purposes

Paging

- Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. In this approach, the operating system divides a process's address space into equal-sized blocks, known as pages, which are mapped to corresponding frames in physical memory. This allows different parts of a program to be stored in various locations within the RAM,

rather than requiring a single continuous space. As a result, paging enhances memory utilization and reduces fragmentation, enabling more efficient and flexible use of system resources.

Segmentation

- In segmentation, memory is divided not into fixed-size units, but based on the program's logical structure. Each segment represents a logical unit such as a module, function, or data structure, and is identified by a name or number. Every segment has both a base address and a limit, which define its starting location and size in memory. This method allows different program modules or data structures to be stored independently, providing better protection, sharing, and dynamic growth for each individual segment. Unlike paging, which uses a page table, segmentation employs a segment table for address mapping. Moreover, data in segmentation is stored contiguously within each segment, reflecting the logical organization of the program.

Translation Lookaside Buffer (TLB)

- The **Translation Lookaside Buffer (TLB)** is a hardware cache that speeds up memory access by storing recent translations of virtual addresses to physical addresses. When a program generates a virtual address, the CPU first checks the TLB for a matching translation. If found, the physical address is quickly retrieved, resulting in faster access. If not, a **TLB miss** occurs, and the CPU refers to the page table in main memory, which takes longer. If the page isn't in memory, the operating system handles a **page fault** by loading it into RAM and updating the TLB. In short, the TLB acts like the CPU's short-term memory, remembering where data is stored to make access faster and reduce delays.

Effective Memory Access Time (EMAT)

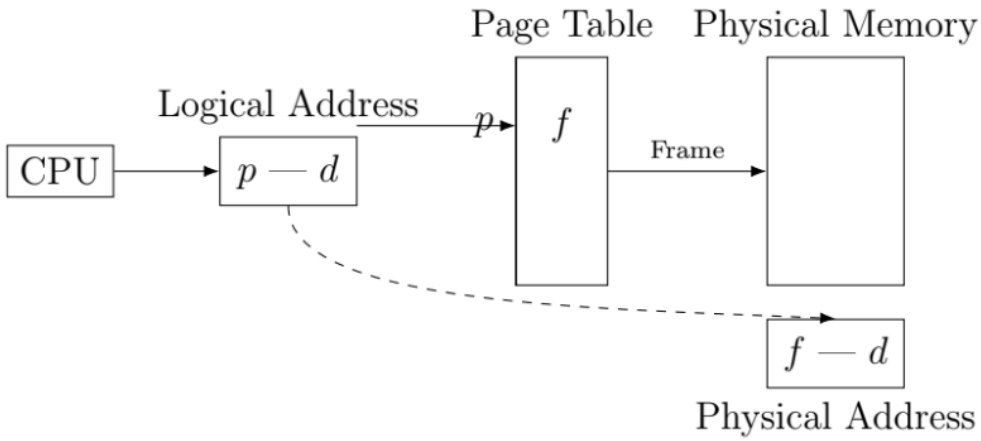
- The **Effective Memory Access Time (EMAT)** represents the average time required to access data or an instruction from memory. It takes into account both the fast and slow scenarios that may occur during memory access, such as Translation Lookaside Buffer (TLB) hits, TLB misses, and potential page faults. By considering the probability and time cost of each of these events, EMAT provides a more accurate measure of the overall speed and efficiency of memory access within a system.

Formula for EMAT

$$\mathbf{EMAT = h \times (T_{tlb} + T_{mem}) + (1-h) \times (T_{tlb} + 2T_{mem})}$$

h: probability of TLB to hit or hit ratio
T_{tlb}: TLB access time
T_{mem}: main memory access time
1-h: probability of TLB miss or miss ratio

Given Scenario



Given: Tmem = 100ns, Ttlb = 10ns, Hit Rate=80%.
Given the all three value we can calculate the EMAT

EMAT = 0.8 x (10ns + 100ns) + (1 - 0.8) x [10ns + 2(100ns)]
= (0.8 x 110) + (0.2)(210)
= 88+42
= 130ns

The average speed to access memory is around 130 nanoseconds.

C. CPU Scheduling Principles

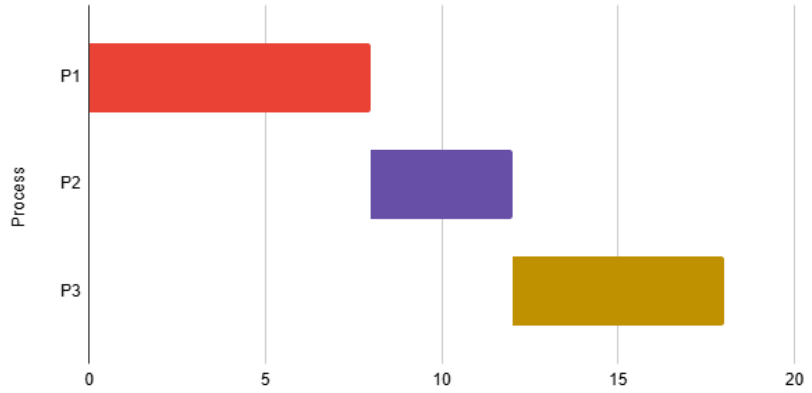
FCFS(First-Come, First-Served) Scheduling

- This scheduling method prioritizes the process that arrives first in the ready queue, allowing it to be executed first. Once a process begins execution, it cannot be interrupted until it finishes. Although this type of scheduling is easy to implement, its non-preemptive nature can lead to inefficiency. If the first process takes much longer than the following ones, it can cause delays and increase the average waiting time causing starvation of other processes, especially in systems with processes of varying lengths.

Round robin scheduling

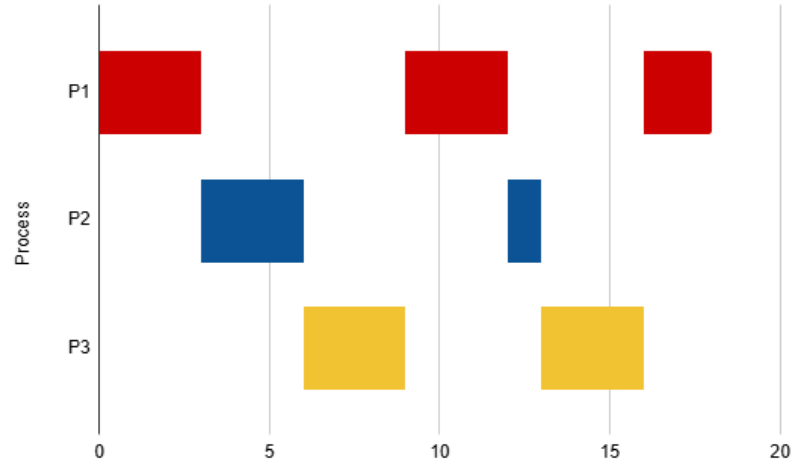
- This method differs from FCFS in that process completion does not solely depend on its length. Instead, it uses a fixed time slice, or quantum, to divide CPU execution time among processes. When a process's time quantum expires, the CPU switches to the next process in the queue, repeating this cycle until all processes are completed. This approach makes it ideal for time-sharing systems and supports effective multitasking. It also helps reduce starvation, as each process is given an equal share of CPU time.

Gannt Chart for P1(8ms), P2(4ms), P3(6ms) FCFS



Sum of waiting time : 0 + 8 + 12 =20 ms
Average waiting time : 20/3= 6.667 ms

Gannt Chart for P1(8ms), P2(4ms), P3(6ms) Quantum time (3ms)



Sum of waiting time : 10 + 9 + 10 =29 ms
Average waiting time : 29/3= 9.667 ms

D. Classic Synchronization Problems

The Dining Philosophers Problem

- The Dining Philosophers problem is a classic synchronization challenge in operating systems that illustrates issues such as process synchronization, deadlock, and resource sharing. In this scenario, five philosophers sit around a circular table with one fork between each pair, making a total of five forks. Each philosopher alternates between thinking and eating, but to eat, they must hold two forks. If all philosophers pick up their left fork first, they will each be waiting for the right fork, leading to a deadlock where no one can eat. To solve this, several strategies can be applied: odd-numbered philosophers may pick up the left fork first while even-numbered ones pick up the right, breaking the circular wait; a moderator can manage fork access and grant permission to prevent conflicts; or they can limit the number of philosophers eating at once to four, ensuring at least one pair of forks remains available

The Producer-Consumer Problem

- The Bounded Buffer Problem is a classic synchronization issue in operating systems. It involves a producer and a consumer sharing a fixed-size memory space called a buffer. The producer generates data items and places them into the buffer, while the consumer removes and processes them. The main challenge is to allow both processes to work concurrently without causing data inconsistency. Problems occur when the producer tries to add data to a full buffer or when the consumer attempts to remove data from an empty one.

  To fix this problem, **semaphores** are used as synchronization tools to control access to shared resources and prevent race conditions. Three semaphores are typically used in this problem: **mutex**, which ensures mutual exclusion so only one process accesses the buffer at a time; **empty**, which tracks available buffer slots to prevent the producer from adding data when the buffer is full; and **full**, which tracks filled slots to prevent the consumer from removing data when the buffer is empty.

The Reader - Writer Problem

- This problem shows how to handle multiple processes that can safely access shared data at the same time. The scenario is that the reader can only read the data and cannot make changes to it, while the writer can modify or update the data. The goal is to allow multiple readers to read and share the data simultaneously without conflict, while ensuring that when the writer edits the data,

only the writer can access it to avoid inconsistency. This problem is similar to a real-life scenario where students can read a shared file, but only the teacher can update its contents. If the teacher edits the file while a student is reading it, it may cause incomplete or incorrect information to be read.

To fix this problem, prioritization can be applied. In the first approach, readers are given priority — when one reader starts, other readers can also read, but the writer must wait until all readers have finished before editing. However, this may cause the writer to starve since it could wait indefinitely for readers to finish. In the second approach, the writer is given priority — no reader can read until the writer has finished editing, which can cause readers to starve while waiting. Both scenarios can lead to deadlock. The proper approach is to balance the priority between readers and writers, allowing them to take turns rather than waiting indefinitely, which prevents deadlock and starvation.