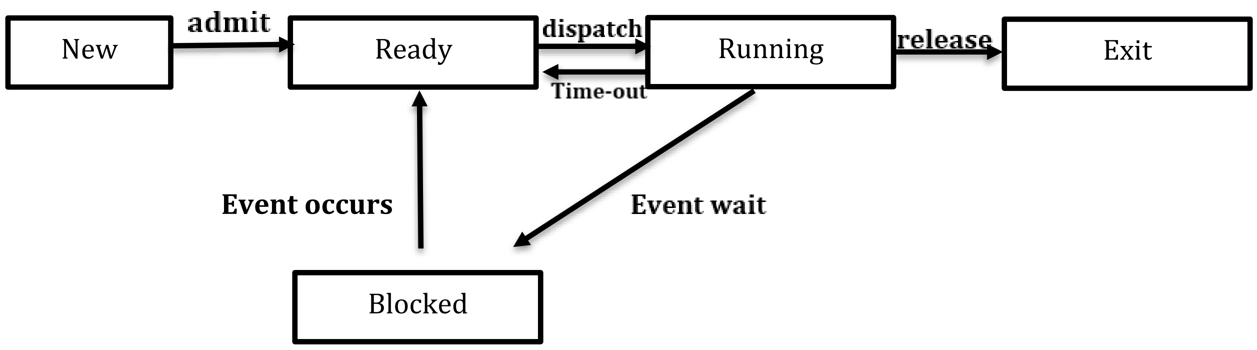
Layer 1: Classic OS Problems Analysis

Process	Threads
A process is an independent program in execution that has its own memory space and resources.	A thread is a smaller unit within a process that shares the same memory and resources with other threads in that process.
Every process has its own memory spaces.	It shares the same memory spaces of the process that it belongs to.
Communication between processes is kind of a bit slower a complex	Can easily communicate with each other since it shares the same memory.
Processes have higher overhead because creating and managing them requires more system resources.	It does have a lower overhead and are faster to create and manage.
If one process crashes/fails, it usually doesn't affect other processes.	If one thread crashes, it can cause the whole process to fail.
Running multiple programs like a web browser and a text editor at the same time.	Opening multiple tabs in a web browser each tab runs as a separate thread within the same process.

Process vs Threads

Based on what I research during the past week, in simple term or how I understand, Process is like a standalone program running on your computer, with its own memory and resources. Meanwhile, a thread is like a smaller task running inside that process. Threads share the same memory, so they can work together more easily, but that also means if one thread fails, it can affect the whole process. Processes are more isolated but heavier to manage, while threads are lighter and faster but riskier when something goes wrong.

The five-state model:



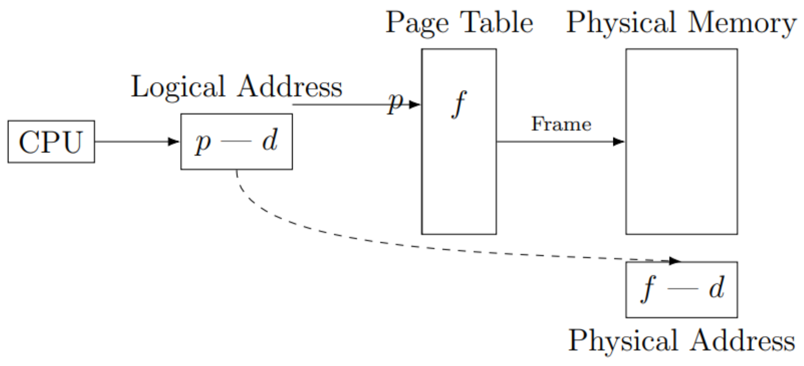
A process goes through different states while it is running. When it is first created, it's in the *New state*, it hasn't started yet but already has a process control block that stores its details. Once it's ready to use the CPU, it moves to the *Ready state*, waiting for its turn to run. When the CPU starts executing it, the process is in the *Running state*, and only one process can run at a time on a single CPU. If the process needs to wait for something, like reading data from a disk, it enters the *Blocked* or *Waiting state*. Finally, when it finishes or is stopped by the user, it moves to the *Exit or Terminate state*, where the operating system removes it from memory.

There are two Inter-Process Communication (IPC) methods, namely Pipes and Message queue so in Pipes it is often use for simple or direct communication between processes for example, in a music streaming app like apple music or spotify (any), one process might handle downloading or buffering the audio data, while another process takes care of decoding and playing the sound. The downloaded data can be sent from one process to the other through a pipe, allowing smooth and continuous music playback while streaming. Meanwhile, the Message queue is useful when several processes need to communicate in an organized way. For example, in a rider app like angkas, when a customer books a ride, the booking details are placed in a message queue. The system then sends the request to nearby riders, and once a rider accepts, updates are sent back through the queue so both the rider and customer get real-time status updates. This ensures smooth and reliable communication between all parts of the system.

Part B: Memory Management Basics

Paging and Segmentation are two ways to manage memory, but they work differently. In paging, memory is divided into fixed-size blocks called pages and frames, which makes it efficient and helps avoid fragmentation. In segmentation, memory is divided into variable-sized segments based on parts of a program like code, data, or stack. This makes it more organized and easier to manage logically, but it can lead to fragmentation since the segments are not all the same size. In short, paging focuses on efficiency, while segmentation focuses on organization.

The Translation Lookaside Buffer (TLB) is a small, fast memory in the CPU that stores recent address translations from virtual to physical memory. When the CPU finds the needed address in the TLB (a TLB hit), access is quick. If not (a TLB miss), it takes longer because the CPU must check the page table. The TLB improves performance by speeding up memory access and reducing delays in address translation.



Effective Memory Access Time (EMAT) Calculation

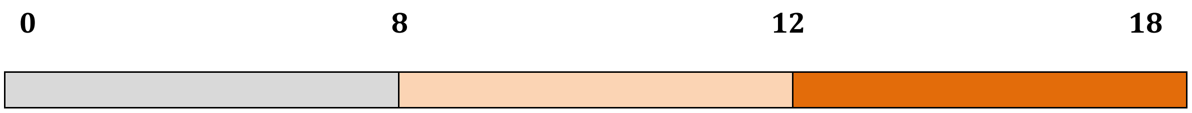
Given: T_{mem} = 100ns, T_{tlb} = 10ns, Hit Rate=80%(0.8)
On tlb hit:
$T_{hit} = T_{tlb} + T_{mem}$ =10 +100 = 10ns
$T_{miss} = T_{tlb} + 2\,(T_{mem})$ = 10+2 (100) = 210ns
EMAT =(0.8)(110)+(0.2)(210) = <u>130ns</u>

Part C: CPU Scheduling Principles

First-Come, First-Served (FCFS) and Round Robin (RR) are both CPU scheduling algorithms but differ in how they handle processes. In FCFS, the process that arrives first is executed first, which is simple but can cause long waiting times if one process runs too long. Meanwhile, Round Robin gives each process a fixed time slice, called a time quantum, before moving to the next process in the queue. This makes it fairer and better for multitasking since every process gets a turn, but it also adds some overhead due to frequent context switching.

Gantt Chart and Average Waiting Time

Process	Duration	Order	Arrival Time
P1	8	1	0
P2	4	2	0
P3	6	3	0



Waiting Times:
P1: 0 ms
P2: 8 ms
P3: 12 ms

Average Waiting Time:
(0+8+12)/3=6.67ms

Round Robin (quantum = 3 ms)

Process	Duration	Arrival Time
P1	8	0
P2	4	0
P3	6	0

P1(0-3)	P2(3-6)	P3(6-9)	P1(9-12)	P2(12-13)	P3(13-16)	P1(16-18)

Part D: Classic Synchronization Problems

The Dining Philosophers Problem shows how hard it is for processes to share limited resources without conflict. As what I have learned during our class in operating system this is the most interesting one because in this setup, five philosophers sit around a table, and each one needs two forks to eat one on the left and one on the right. The problem starts when every philosopher picks up one fork at the same time and waits for the other, causing a deadlock, where no one can continue. Another issue is starvation, which happens when one philosopher never gets a chance to eat because others keep getting the forks first.

Some common solutions include using semaphores or monitors to control when a philosopher can pick up forks, numbering the forks so each philosopher always picks them up in a set order, limiting the number of philosophers who can try to eat at once, or making them follow different rules (for example, some pick up the left fork first, others the right). This problem helps us understand how operating systems manage multiple processes that need shared resources without causing system freezes or unfair waiting.

The Producer-Consumer problem is a classic synchronization challenge where one or more producer processes generate items and place them in a fixed-size buffer, while one or more consumer processes remove items from that buffer. The key issues are: the producer must not add an item when the buffer is full, and the consumer must not remove an item when the buffer is empty also, they must not both try to access the buffer at the same time so as to avoid data corruption. To coordinate all this, we use semaphores, which are integer counters with two main atomic operations: wait(S) (which decreases the value and possibly blocks if it becomes non-positive) and signal(S) (which increases the value and may wake up a blocked process)

The Reader-Writer Problem happens when several processes need to share a resource, like a file or database. Readers only view the data, while writers update it. Multiple readers can access the resource at once since they don’t change anything, but only one writer should work at a time to avoid data errors. There are two common versions: one that prioritizes readers (which can make writers wait too long) and another that prioritizes writers (which can delay readers). Semaphores are used to manage access and keep everything synchronized and fair.