The **Operating System** works as an *intermediary* between the user and the hardware. OS manages the hardware such as the **CPU, Main Memory & I/O devices.**

Which allows the OS to **LOAD** program executables, initialize the PC & other registers, and allows the OS to **READ & WRITE** files from disk.

Process Abstraction: When you run an exec file, the OS creates a process == a running program. A **process** has **PID, Code & Data (Static) & Stack and Heap.**

States of Process: Running (executing), Ready(waiting to be scheduled) , Blocked (waiting for some event), New (Being Created) & Dead/Zombie (terminated)

System Call: is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.

fork() creates a new child process | **exec()** makes a process execute a given executable | **exit()** terminates the process | **wait()** blocks parent until child is done

Multiprogramming - A computer running more than one process at a time | **Multiprocessing:** A computer using more than 1 CPU at a time | **Multitasking**: Tasks sharing a common resource at a time | **Multithreading** - is an extension of multitasking

Trap Instruction: It is a software interrupt generated by the user program or by an error when the **operating system** is needed by it to perform the system calls

USER MODE (NON-Privileged): Ex. Reading the status of processor, Reading the system time, generate any trap, sending final printout of printer.

Kernel Mode (Privileged): I/O instructions and halt instructions, turn off all interrupts, set the timer, context switching, clear memory, remove process from mem.

I/O Protection - CPU Protection - Main Memory Protection: To prevent the access of unauthorized users and To ensure that each active programs or processes in the system uses resources only as the stated policy, To improve reliability by detecting latent errors.

Programmed I/O (Busy Looping): Usually the transfer is from a CPU register and memory. In this case it requires constant monitoring by the CPU of the peripheral devices.

Interrupt Driven: Upon detection of an external interrupt signal the CPU stops momentarily the task that it was already performing, branches to the service program to process the I/O transfer, and then return to the task it was originally performed.

DMA: The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

Interprocess Communication: Is a mechanism which allows processes to communicate w/ each other & synchronize actions. Shared Memory and Message Passing

Race Conditions: It is possible to have a software system, in which the *output depends on the sequence of events.* If the event does not occur as the software developer wanted, a fault happens. This fault is a RC.. (RC takes place when multiple processes or threads operate on a shared data | RC is studied under conconcurrent processes)

Process Synchronization: Is where we allow only one process to enter and manipulates the shared data in Critical Section. (This is how we solve the Race Condition)

Bounded Buffer: A **bounded buffer** lets multiple producers and multiple consumers share a single **buffer**.

Bounded Buffer Problem: The Producer (P) Consumer(C) problem says that a P produces items 1 by 1 and places them in a buffer, and a C takes items from the buffer 1 by 1. If buffer is full, the P doesn't try to place more items, but waits for an empty space. If the buffer is empty, the C doesn't try to take an item but just waits for a new item to appear.

ReaderWriter Prob: One set of data is shared among a number of processes. Once a writer is ready, it performs its write. Only one writer may write at a time. If a process is writing, no other process can read it . If at least one reader is reading, no other process can write. Readers may not write and only read. Perfect solution (WRITER__wait (wrmutex); write here; signal(wrmutex)__READER (wait(mutex) reader ++/--if reader == 1/0(wait wrmutex) (LOCK)(Critical Point) signal(mutex) READ HERE (Repeat same but w/ 0 & ***Reader - Preference Policy

Dining Philosophers Prob: the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. Solution (Waiter to monitor, use lock)

Peterson Algorithm: The algorithm uses two variables, *flag and turn.* A flag[n] value of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.

Context switching: involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

Critical Section: Is a code segment that accesses shared variables and has to be executed as an atomic action.

Locks - are methods of synchronization used to prevent multiple threads from accessing a resource at the same time.

Mutex: Mutex is a mutual exclusion object that synchronizes access to a resource. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section.

Semaphore: is a variable or abstract data type *used to control access* to a common resource by multiple processes in a concurrent **system** such as a multitasking **operating system**

***A **Mutex** is **different** than a **semaphore** as it is a locking mechanism while a **semaphore** is a signalling mechanism.

Wait(): Wait is called when a process wants access to a resource

Signal(): Is called when a process is done using a resource.

Disadvantages of Semaphore: BUSY WAITING: continual looping of semaphore process. && **SpinLock**: When busy waiting happens and wastes CPU cycle

Mutual exclusion no two process can be simultaneously present inside critical section at any point of time, only one process can enter into critical section at any point of time.

Progress: No process running outside the critical section should block the other interested process from entering into it's critical section when in fact the critical section is free.

Bounded wait: No process should have to wait forever to enter into critical section. there should be boundaries on getting chances to enter into critical section. If bounded waiting is not satisfied then there is a possibility of starvation. *** You can put bound on process but not on time spent in critical point

Fairness: The solution to starvation is called "**fairness**" - that all threads are fairly granted a chance to execute

Starvation: occurs when a low priority program is requesting for a **system** resource, but are not able to execute because a higher priority program is utilizing that resource for an extended period.

Deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states but incase of **Non-deterministic algorithm**, for the same input, the compiler may produce different outputs in different runs.

Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance.

Deadlock: Is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process

***Note there is *no constraint* on whether ProcessX enters C or ProcessY enter C | ProcessX and ProcessY are *concurrent*

Generally speaking there are three ways of handling deadlocks: **Deadlock prevention or avoidance** - Do not allow the system to get into a deadlocked state. **Deadlock detection and recovery** - Abort a process or preempt some resources when deadlocks are detected. **Ignore the problem all together** - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take. The **process** with the lowest **deadlock** priority is set as **deadlock victim**.

Safety Condition: A state is safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.

Optimistic algorithm performs better bc resources are allocated right away if resources are available. However for a **Pessimistic Approach** there is more emphasis on security so resources are not always allocated right away

Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.

- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)

- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.

- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

The banker's algorithm relies on <u>several key data structures</u>: (where n is the number of processes and m is the number of resource categories.)

Available[m] indicates how many resources are currently available of each type.

Max[n][m] indicates the maximum demand of each process of each resource.

Allocation[n][m] indicates the number of each resource category allocated to each process.

Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)

For simplification of discussions, we make the following notations / observations:

One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.

A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i

Safety Algorithm This algorithm determines if the current state of a system is safe, according to the following steps:

Let Work and Finish be vectors of length m and n respectively.

Work is a working copy of the available resources, which will be modified during the analysis.

Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)

Initialize Work to Available, and Finish to false for all elements.

Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.

Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.

If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

Resource-Request Algorithm (The Banker's Algorithm) :When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:

Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.

If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.

Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:

Available = Available - Request

Allocation = Allocation + Request

Need = Need - Request