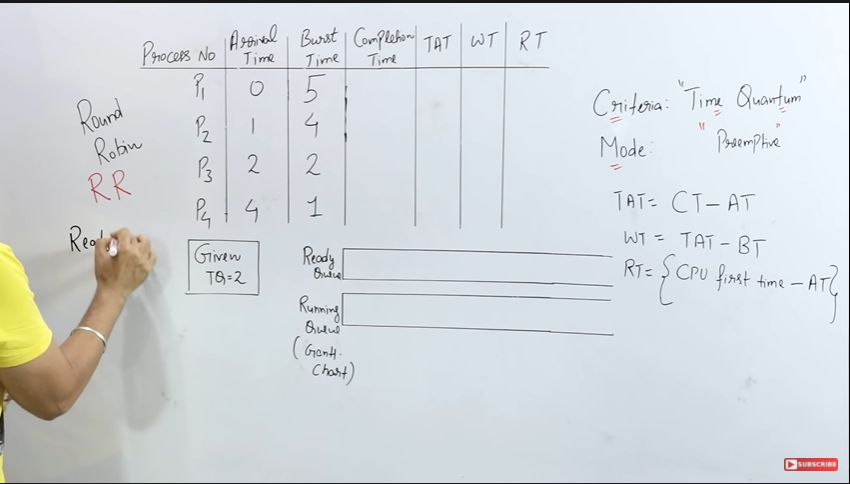
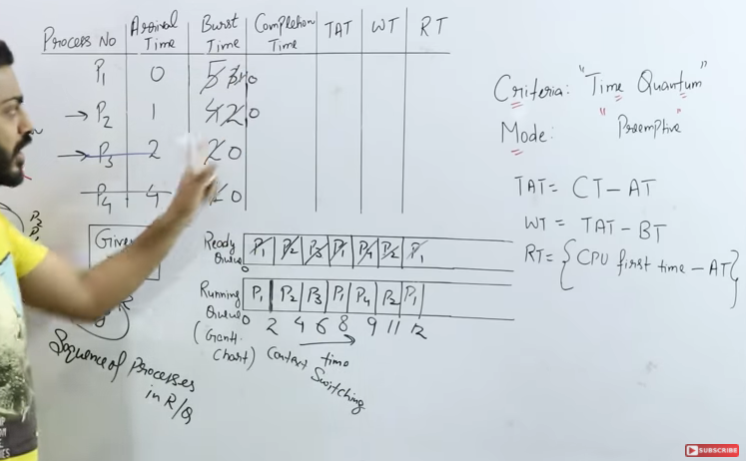
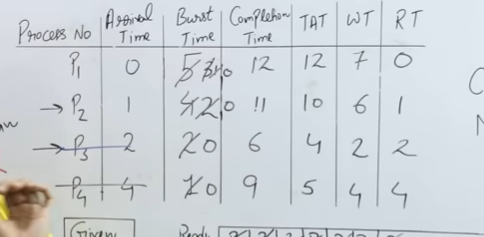
## Round Robin







## Multilevel Queue

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

**For example,** A common division is made between foreground(or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A multi-level queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

## **Advantages of Multilevel Queue Scheduling**

With the help of this scheduling we can apply various kind of scheduling for different kind of processes:

**For System Processes**: First Come First Serve(FCFS) Scheduling.

**For Interactive Processes**: Shortest Job First (SJF) Scheduling.

**For Batch Processes**: Round Robin(RR) Scheduling

**For Student Processes**: Priority Scheduling

## **Disadvantages of Multilevel Queue Scheduling**

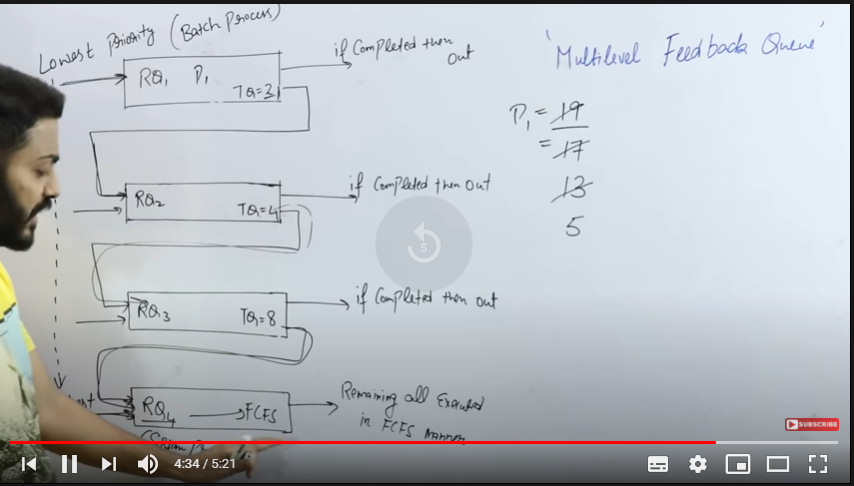
The main disadvantage of Multilevel Queue Scheduling is the problem of starvation for lower-level processes.

Let us understand **what is starvation?**

**Starvation:**

Due to starvation lower-level processes either never execute or have to wait for a long amount of time because of lower priority or higher priority process taking a large amount of time.

## Multilevel Feedback Queue (Overcoming Starvation)



## Multi-Processor Scheduling

In multiple-processor scheduling **multiple CPU’s** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

**Approaches to Multiple-Processor Scheduling**

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

**Processor Affinity –**

Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.  
When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **PROCESSOR AFFINITY**.

There are two types of processor affinity:

1. **Soft Affinity –** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity –** Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like *sched\_setaffinity()* that supports hard affinity.

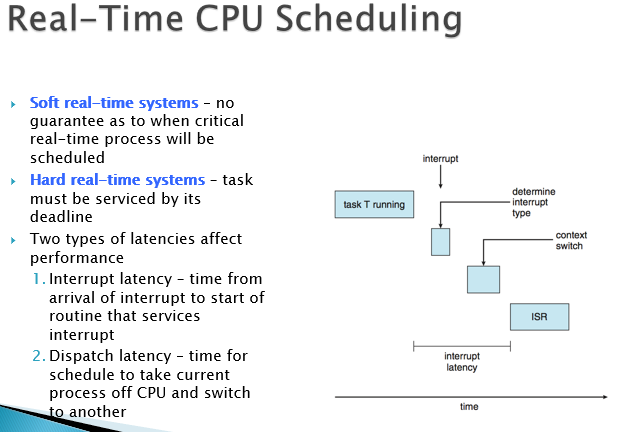
**Load Balancing –**

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing :

1. **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

## Real Time Cpu Scheduling



# Threads

Thread is a basic unit of CPU utilization.

It comprises of

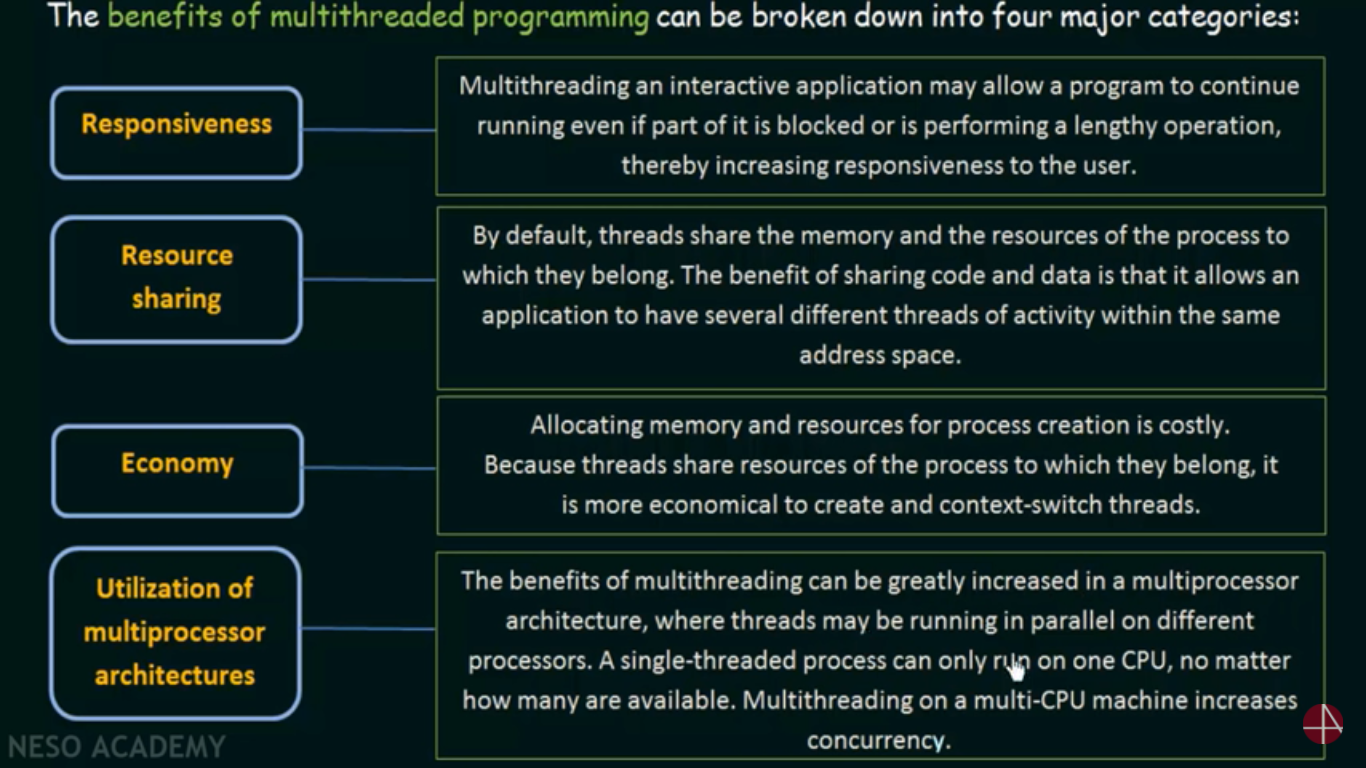
1. Thread ID
2. A Program Counter
3. Register Set
4. Stack

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional process has a single thread of control.

If it has multiple threads of control, it can perform more than one task at a time.

## Benefits of Multithreading



## Types of Threads

### User Threads

Supported above the kernel and are managed without the support of the Kernel.

### Kernel Threads

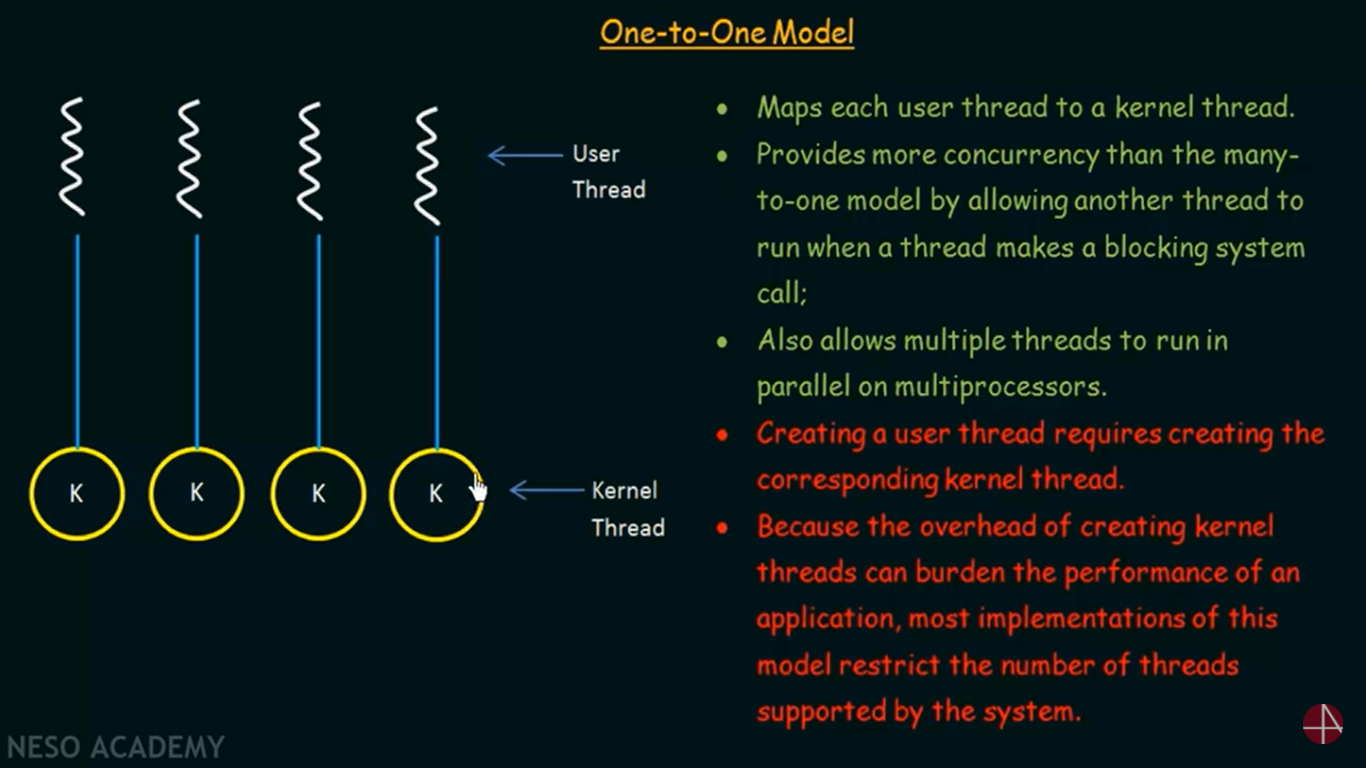
Supported and Managed directly by the Operating System.

## Thread Models

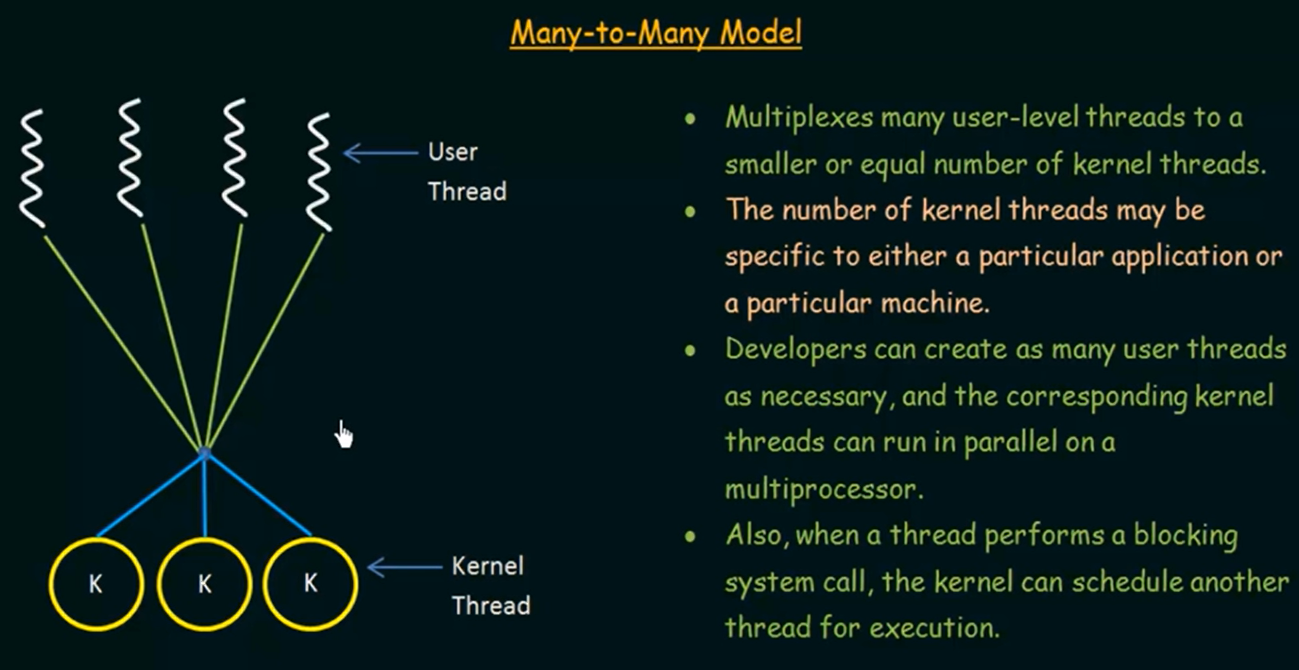
### Many-to-One Model



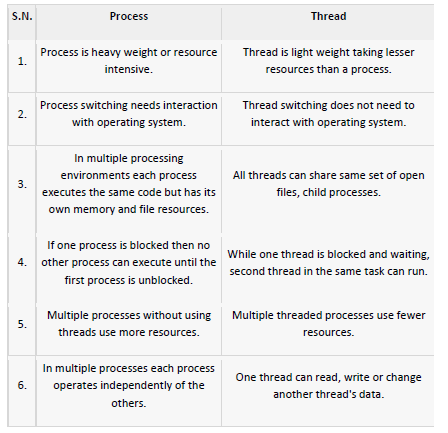
### One-To-One Model



### Many-To-Many Model



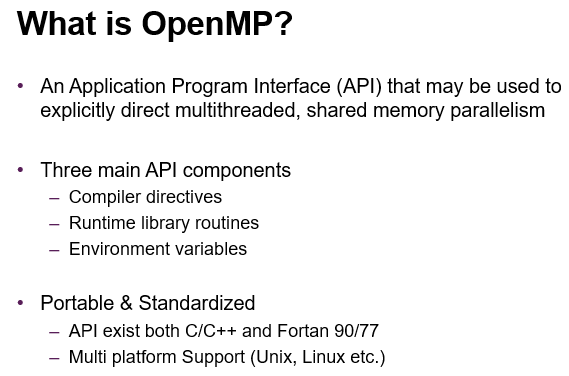
## Process Vs. Thread



## Thread States

1. Spawn 🡪 One Threads spawn another thread within the process.
2. Blocked 🡪 Threads is waiting for an event.
3. Unblocked 🡪 The event had occurred.
4. Finished 🡪 The thread is done doing the task it was assigned to do.

## OpenMP



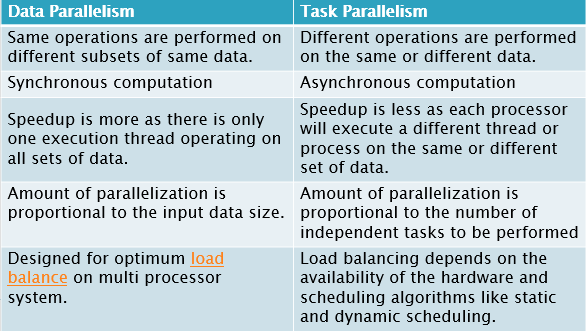
## Types of Parallelism

### Data Parallelism

Focus on distributing data across different parallel computing nodes.

### Task Parallelism

Focus on distributing threads across different parallel computing nodes.



## Amdahl’s Law

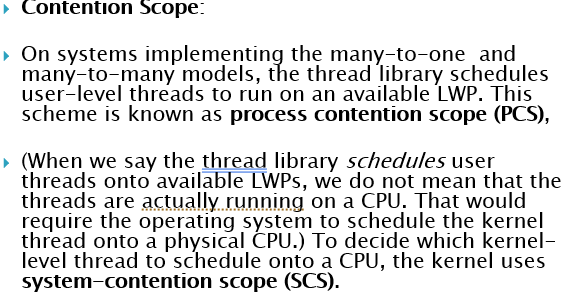


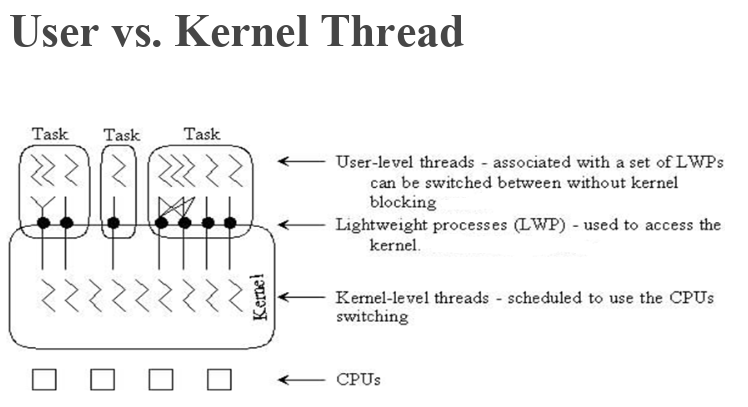
S 🡪 Portion of program executed serially.

N 🡪 No of cores.

## Thread Scheduling

### Contention Scope





# Process Synchronization

### Co-operative Process

In this process the execution of one process affects the execution of other.

Two or more process share a common variable.

They can share some memory or buffer.

They can share some common code.

### Independent Process

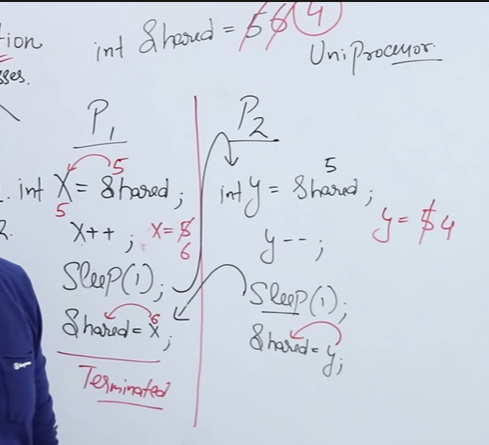
Execution of one process doesn’t effect the other.

## Process Synchronization

It is essential that the co-operative process are synched properly else they might create problems like dreadlocks.

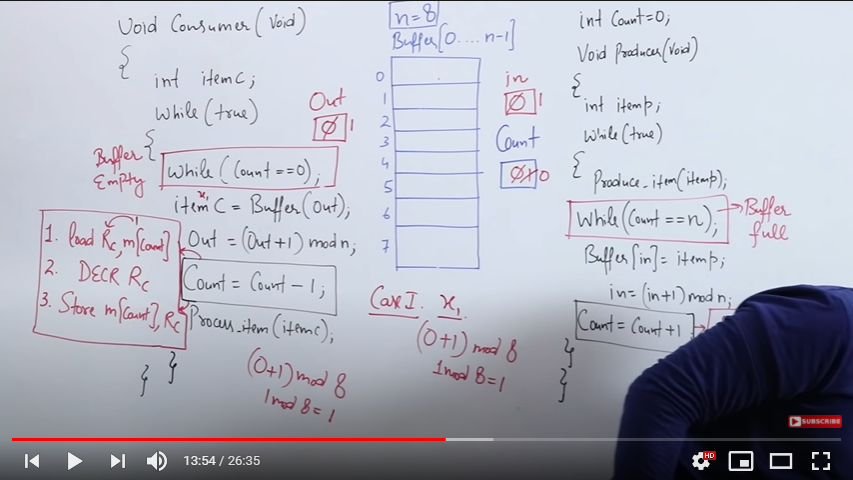
## Race Condition

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

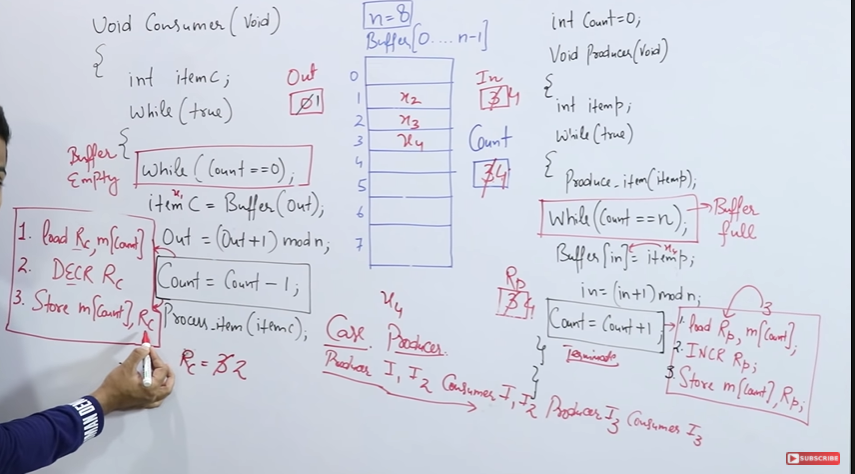


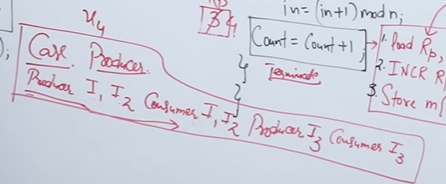
## Producer – Consumer Problem

### Best Case – Perfect Sync



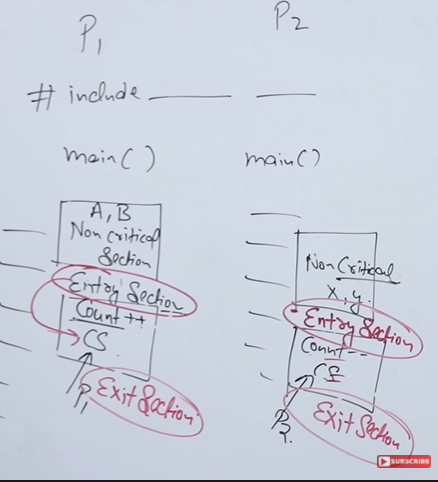
### Second Case – Failed Sync





## Critical Section

It is a part of a program where shared resouces are accessed by varoius programs.



### Conditions required to avoid race condition:

No two processes may be simultaneously inside their critical regions.

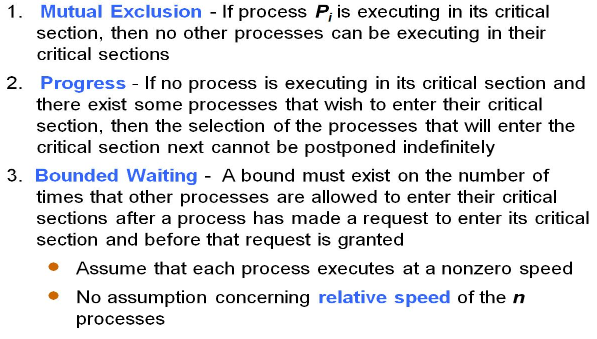
No assumptions may be made about speeds or the number of CPUs.

No process running outside its critical region may block other processes.

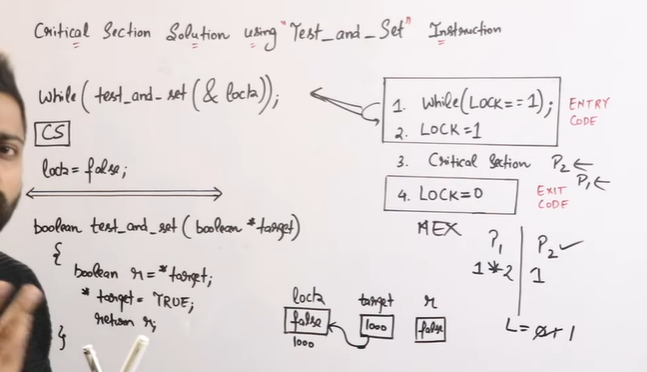
No process should have to wait forever to enter its critical region.

## Sync Mechanism – 4 RUles

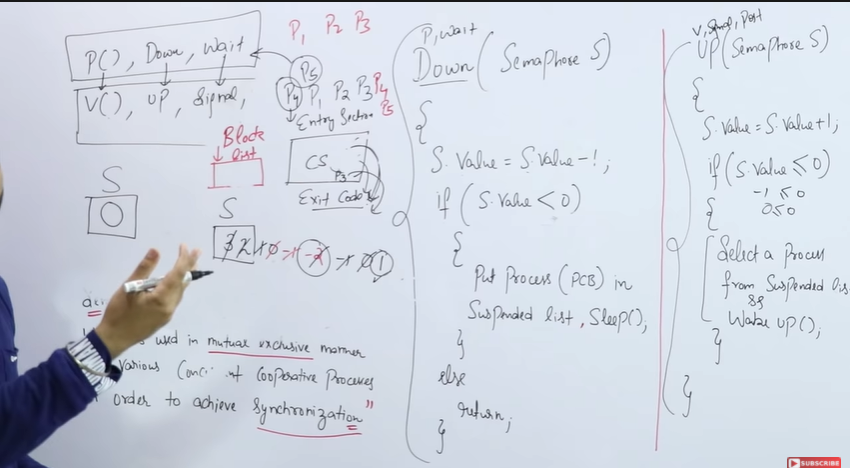
1. Mutual Exclusion.
2. Progress.
3. Bounded Wait.
4. No assumptions related to Hardware speed.



## Test and Set



# Semaphores



S = 0

Means there are 0 processes in the suspended/waiting list

S = -1

There is 1 processes in the suspenede/waiting list.

S = 10

It means, 10 process can enter the Critical section.

