



# Process Management in Operating System (unit2)

By: Mrs. Nidhi Divecha (ME CMPN)

# What is Process Management?

- Process management involves various tasks like **creation, scheduling, termination of processes, and a dead lock**.
- Process is a program that is under execution, which is an important part of modern-day operating systems.
- The OS must allocate resources that enable processes to share and exchange information. It also protects the resources of each process from other methods and allows **synchronization** among processes.
- It is the job of OS to manage all the **running processes** of the system.
- It handles operations by performing tasks like **process scheduling** and such as **resource allocation**.

# Operating System - Processes

- A **process** is basically a **program** in execution. The execution of a process must progress in a **sequential fashion**.
- The OS helps you to create, schedule, and terminates the processes which is used by CPU. A process created by the **main process is called a child process**.
- **Process operations** can be easily controlled with the help of **PCB(Process Control Block)**.
- It as the brain of the process, which contains all the crucial information related to processing like **process id, priority, state, CPU registers, etc.**

There are number of concepts associated with the **process management function** of an operating system. Some of those concepts are given as following.

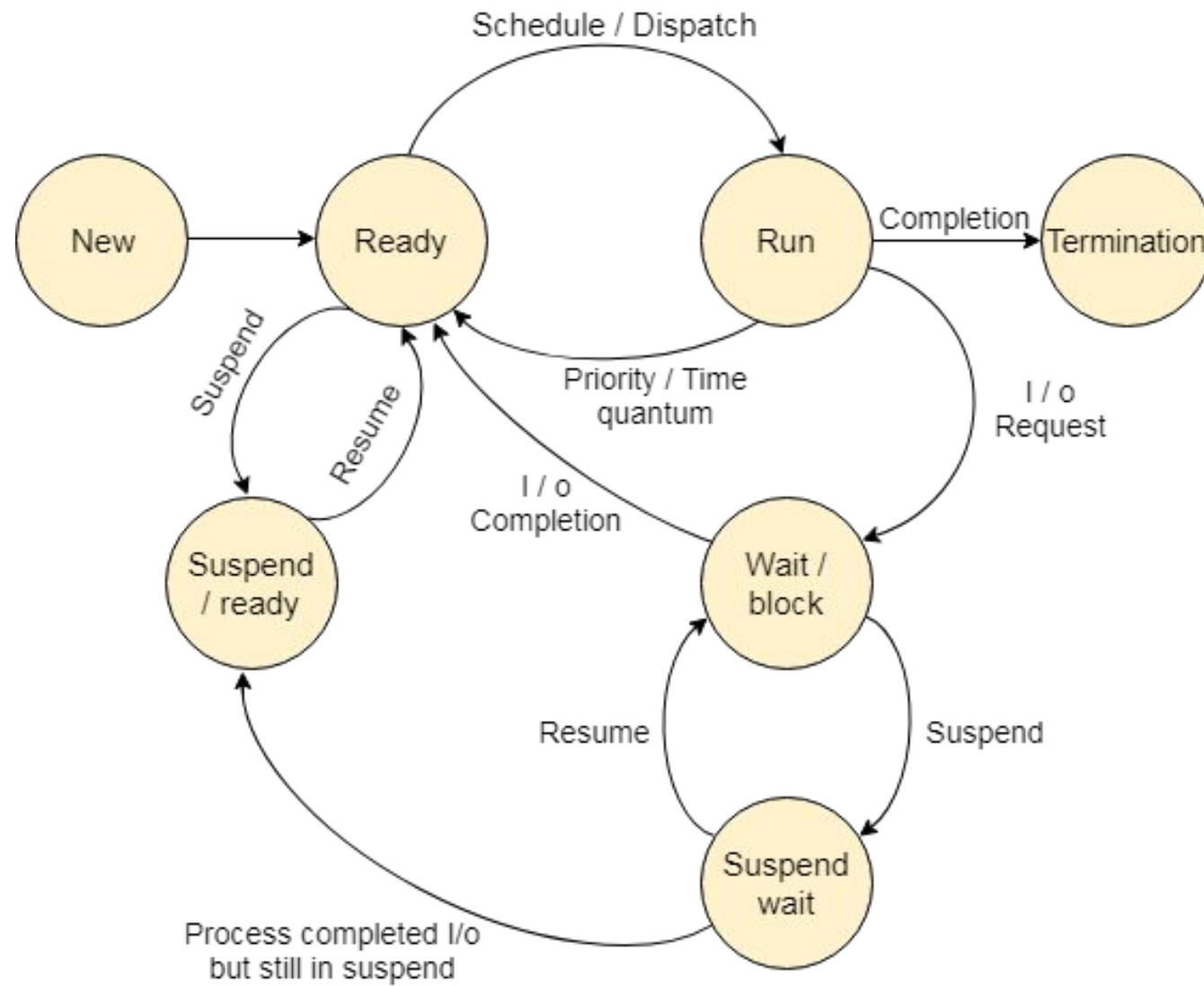
- Process State
- Process Control Block (PCB)
- Process Operations
- Process Scheduling
- Process Synchronization
- Inter process Communication
- Deadlock

# Process Architecture



- **Stack:** The Stack stores temporary data like function parameters, returns addresses, and local variables.
- **Heap** Allocates memory, which may be processed during its run time.
- **Data:** It contains the variable.
- **Text:** Text Section includes the current activity, which is represented by the value of the Program Counter.

# Process States



- The process, from its creation to completion, passes through various states. The minimum number of states is five.
- The names of the states are not standardized although the process may be in one of the following states during execution.

### **1. New**

- A program which is going to be picked up by the OS into the main memory is called a new process.

### **2. Ready**

- Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.
- The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state.

### **3. Running**

- One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have  $n$  processors in the system, then we can have  $n$  processes running simultaneously.

#### **4. Block or wait**

- From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process.
- When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

#### **5. Completion or termination**

- When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

#### **6. Suspend ready**

- A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.
- If the main memory is full and a higher priority process comes for the execution, then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.

#### **7. Suspend wait**

- Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.

# Operations on the Process

## 1. Creation

- Once the process is created, it will be ready and come into the ready queue (main memory) and will be ready for the execution.

## 2. Scheduling

- Out of the many processes present in the ready queue, the Operating system chooses one process and start executing it. Selecting the process which is to be executed next, is known as scheduling.

## 3. Execution

- Once the process is scheduled for the execution, the processor starts executing it. Process may come to the blocked or wait state during the execution then in that case the processor starts executing the other processes.

## 4. Deletion/killing

- Once the purpose of the process gets over then the OS will kill the process. The Context of the process (PCB) will be deleted, and the process gets terminated by the Operating system.



# Operations on the Process

The operations of process carried out by an operating system are primarily of two types:

**Process creation**

**Process termination**

# Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)

## Resource sharing

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

## Execution

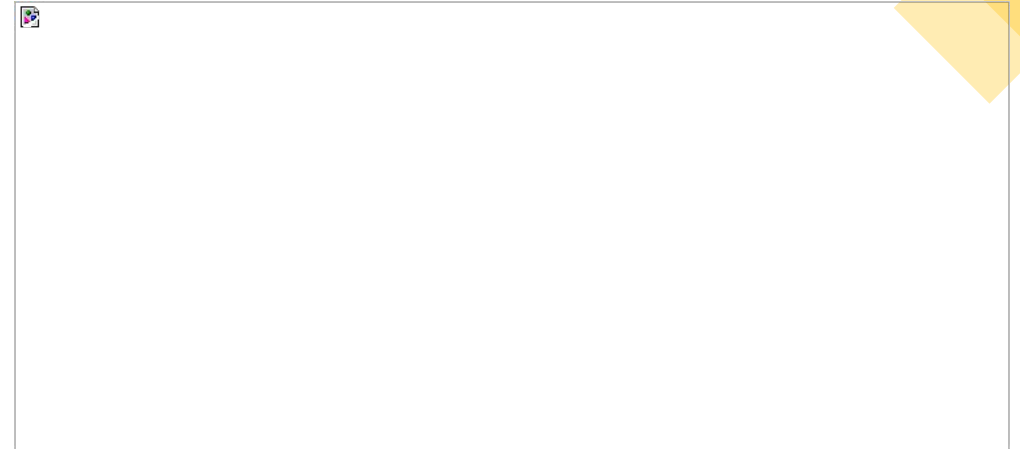
- Parent and children execute concurrently
- Parent waits until children terminate

## Address space

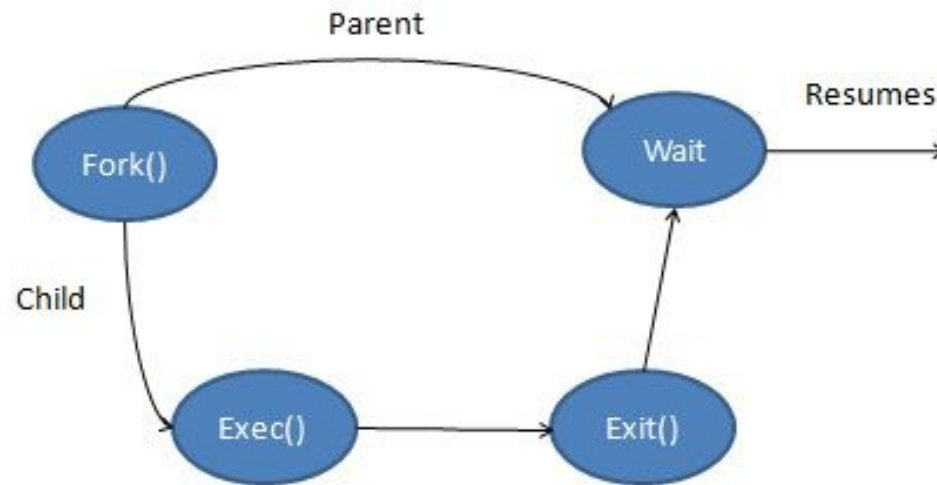
- Child duplicate of parent
- Child has a program loaded into it

## UNIX examples

- fork system call creates new process
- exec system call used after a fork to replace the process' memory space with a new program



- A diagram that demonstrates **process creation using fork()** is as follows –



- If the `fork()` system call returns the **negative value**, the creation of a child process was **unsuccessful**.
- `fork()` returns a **zero** to the **newly created process**.
- `fork()` returns a **positive value**, the **process id** of the child process to the parent.

## 2) Process termination

- Process termination is an operation in which a process is terminated **after the execution of its last instruction.**
- When a process is terminated, the resources that were being utilized by the process are released or deallocated by the operating system.
- A process terminates when it finishes executing its final statement and asks operating system to delete it by using the **exit() system call.**
- When a child process terminates, it sends the status information back to the parent process before terminating via **the wait() system call.**
- The child process can also be terminated by the parent process if the task performed by the child process is no longer needed.
- In addition, when a parent process terminates, it has to terminate the child process as well because a child process cannot run when its parent process has been terminated.
- The termination of a process when all its instruction has been executed successfully is called normal termination. However, there are instances when a process terminates due to some error. This termination is called as abnormal termination of a process.

# Process Control Block(PCB)

- A Process Control Block is a **data structure** maintained by the Operating System for every process.
- The PCB is identified by an **integer process ID (PID)**.
- The PCB is maintained for a process throughout its lifetime and is deleted once the process terminates.
- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems.
- Here, are important components of PCB



**Process Control Block  
(PCB)**

- **Process Id:** A process id is a **unique identity** of a process. Each process is identified with the help of the process id.
- **Process state:** A process can be **new, ready, running, waiting, etc.**
- **Program counter:** The program counter lets you know the **address of the next instruction**, which should be executed for that process.
- **Priority:** There is a priority associated with each process. Based on that priority the CPU finds which process is to be executed first. Higher priority process will be executed first.
- **CPU registers:** During the execution of a process, it deals with a number of data that are being used and changed by the process. But in most of the cases, we have to stop the execution of a process to start another process and after sometimes, the previous process should be resumed once again. Since the previous process was dealing with some data and had changed the data so when the process resumes then it should use that data only. These **data are stored** in some kind of storage units called registers.
- **List of opened files:** A process can deal with a number of files, so the CPU should maintain a list of files that are being opened by a process to make sure that no other process can open the file at the same time.
- **List of I/O devices:** A process may need a number of I/O devices to perform various tasks. So, a proper list should be maintained that shows which I/O device is being used by which process.
- **Protection:**

# Operating System - Process Scheduling

- **Process Scheduling** is an OS task that schedules processes of different states like ready, waiting, and running.
- The act of determining which process is in the **ready** state and should be moved to the **running** state is known as **Process Scheduling**.
- Another important reason for using a process scheduling system is that it keeps the CPU busy all the time.
- This allows you to get the minimum response time for programs.
- The objective of multiprogramming is to have some process running all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process for program execution on the CPU.

# Process Scheduling in OS

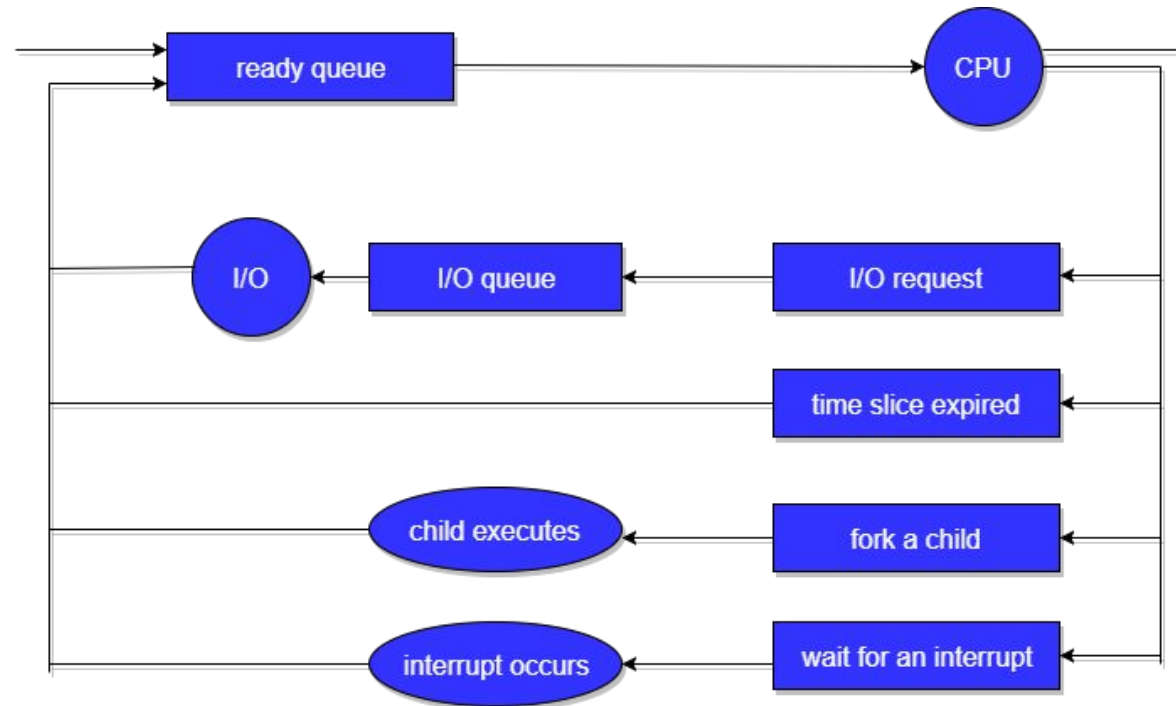
- The act of determining which process in the ready state should be moved to the running state is known as *Process Scheduling*.

## Process Scheduling Queues

- All processes when enters into the system are stored in the **job queue**.
- Processes in the Ready state are placed in the **ready queue**.
- Processes waiting for a device to become available are placed in **device queues**. There are unique device queues for each I/O device available.



- A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, once of several events could occur.
- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



In the above-given Diagram,

- **Rectangle** represents a **queue**
  - **Circle** denotes the **resource**
  - **Arrow** indicates the **flow of the process**.
- 
- Every new process first put in the Ready queue .It waits in the ready queue until it is finally processed for execution. Here, the new process is put in the ready queue and wait until it is selected for execution or it is dispatched.
  - One of the processes is allocated the CPU and it is executing
  - The process should issue an I/O request
  - Then, it should be placed in the I/O queue.
  - The process should create a new subprocess
  - The process should be waiting for its termination.
  - It should remove forcefully from the CPU, as a result interrupt. Once interrupt is completed, it should be sent back to ready queue.

## **Two State Process Model**

- Two-state process models are:
- Running
- Not Running

### **Running**

- In the Operating system, whenever a new process is built, it is entered into the system, which should be running.

### **Not Running**

- The process that are not running are kept in a queue, which is waiting for their turn to execute. Each entry in the queue is a point to a specific process.

## **Type of Process Schedulers**

- A scheduler is a type of system software that allows you to handle process scheduling.

There are mainly three types of Process Schedulers:

- **Long Term**
- **Short Term**
- **Medium Term**

## Long Term Scheduler

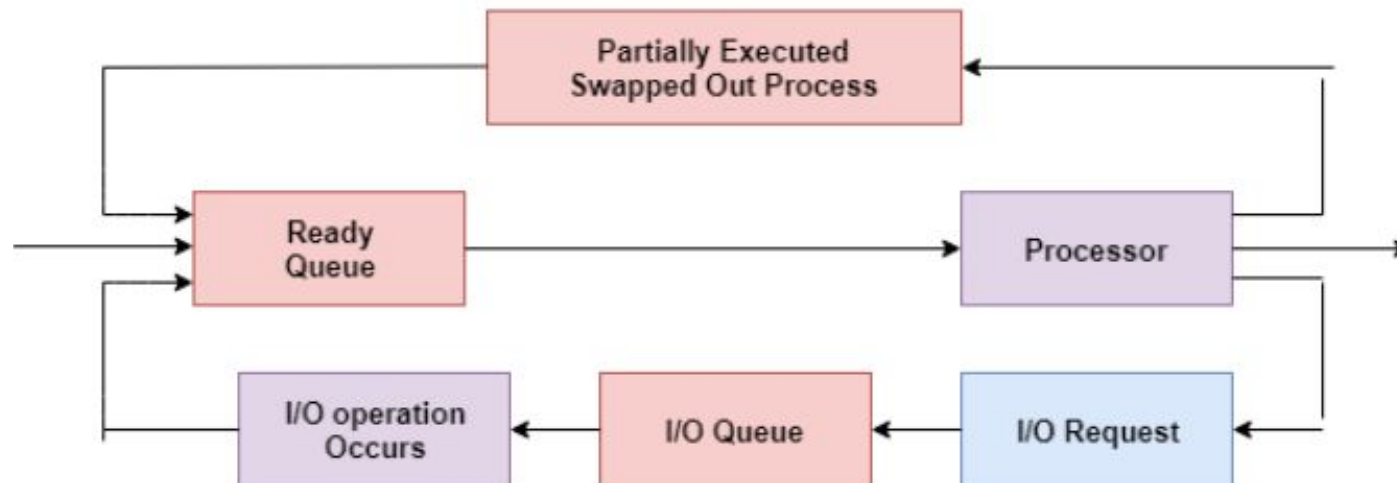
- Long term scheduler is also known as a **job scheduler**.
- long-term scheduler selects processes from the storage pool in the secondary memory and loads them into the ready queue in the main memory for execution.
- The long-term scheduler controls the degree of multiprogramming.
- It must select a careful mixture of I/O bound and CPU bound processes to yield optimum system throughput.
- If it selects too many CPU bound processes, then the I/O devices are idle and if it selects too many I/O bound processes then the processor has nothing to do.
- The job of the long-term scheduler is very important and directly affects the system for a long time.

## Short Term Scheduler

- Also known as **CPU scheduler or dispatcher**.
- The short-term scheduler selects one of the processes from the ready queue and schedules them for execution.
- A scheduling algorithm is used to decide which process will be scheduled for execution next.
- The short-term scheduler executes much more frequently than the long-term scheduler as a process may execute only for a few milliseconds.
- The choices of the short-term scheduler are very important.
- If it selects a process with a long burst time, then all the processes after that will have to wait for a long time in the ready queue.
- This is known as starvation and it may happen if a wrong decision is made by the short-term scheduler.

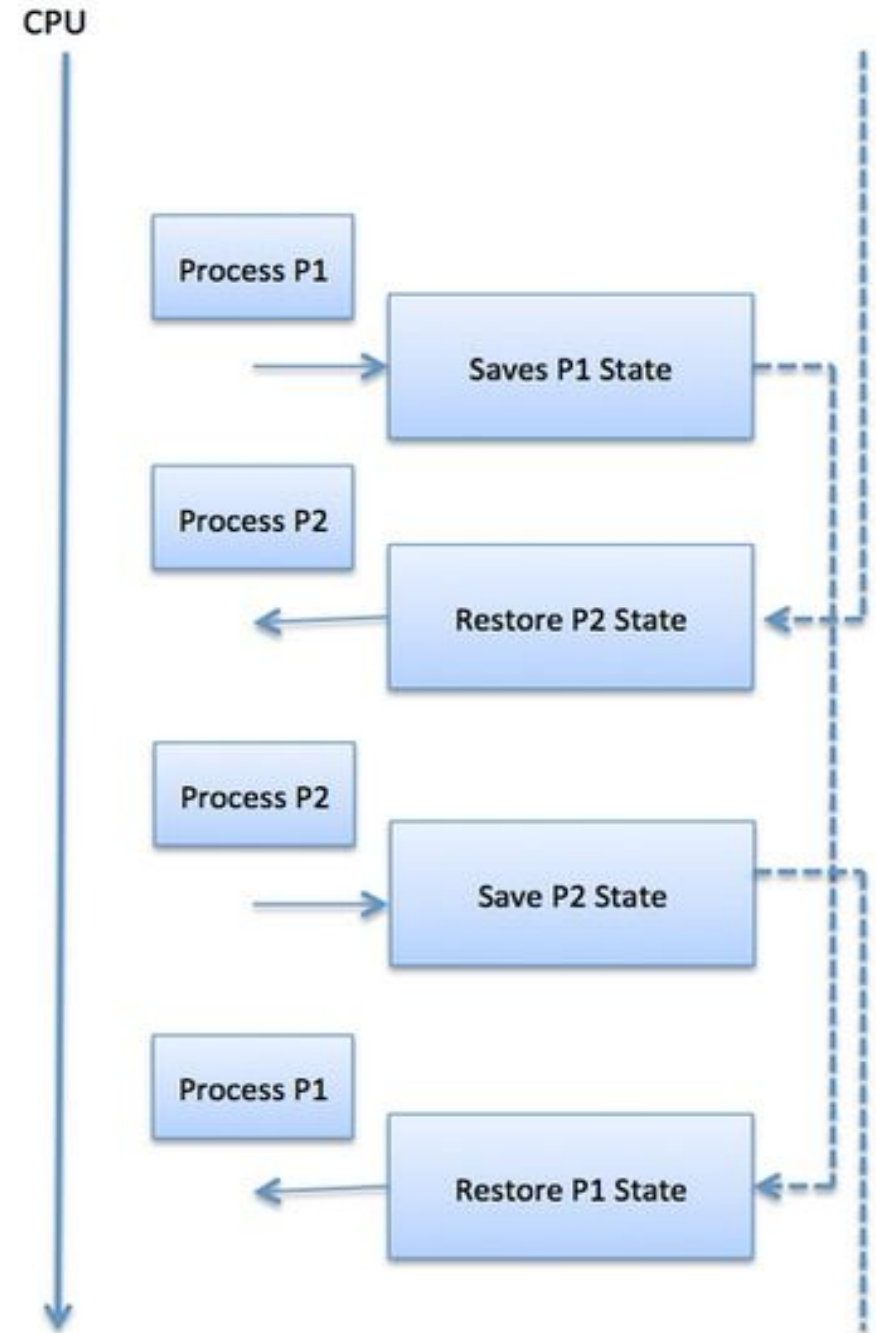
# Medium Term Scheduler

- Medium term scheduler takes care of the **swapped-out** processes.
- If the running state processes needs some IO time for the completion, then there is a need to change its state from running to waiting.
- Medium term scheduler is used for this purpose. It removes the process from the running state to make room for the other processes. Such processes are the swapped-out processes, and this procedure is called **swapping**.
- The medium-term scheduler is responsible for suspending and resuming the processes.



# What is Context switch?

- A **context switch** is the mechanism to **store and restore** the **state or context of a CPU** in Process Control block(PCB) so that a process execution can be resumed from the same point at a later time.
- The Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system.
- When switching perform in the system, it stores the old running process's status in the form of registers and assigns the CPU to a new process to execute its tasks.
- While a new process is running in the system, the previous process must wait in a ready queue.
- The execution of the old process starts at that point where another process stopped it.
- It defines the characteristics of a multitasking operating system in which multiple processes shared the same CPU to perform multiple tasks without the need for additional processors in the system.
- In the above diagram, initially Process 1 is running. Process 1 is switched out and Process 2 is switched in because of an interrupt or a system call. Context switching involves saving the state of Process 1 into PCB1 and loading the state of process 2 from PCB2. After some time again a context switch occurs, and Process 2 is switched out and Process 1 is switched in again. This involves saving the state of Process 2 into PCB2 and loading the state of process 1 from PCB1.



# Process Synchronization

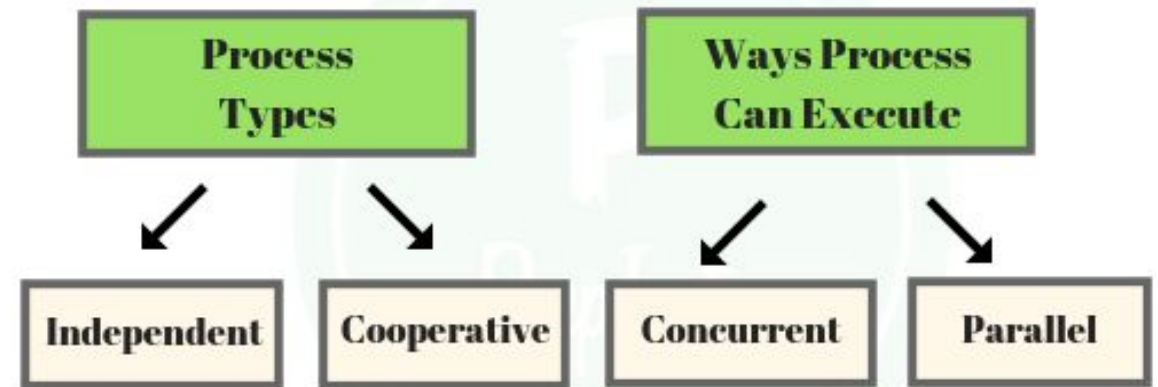
- **Process Synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
- It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.
- This can lead to the inconsistency of shared data.
- To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

Based on synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.
- Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.



## Process Synchronization Types in OS

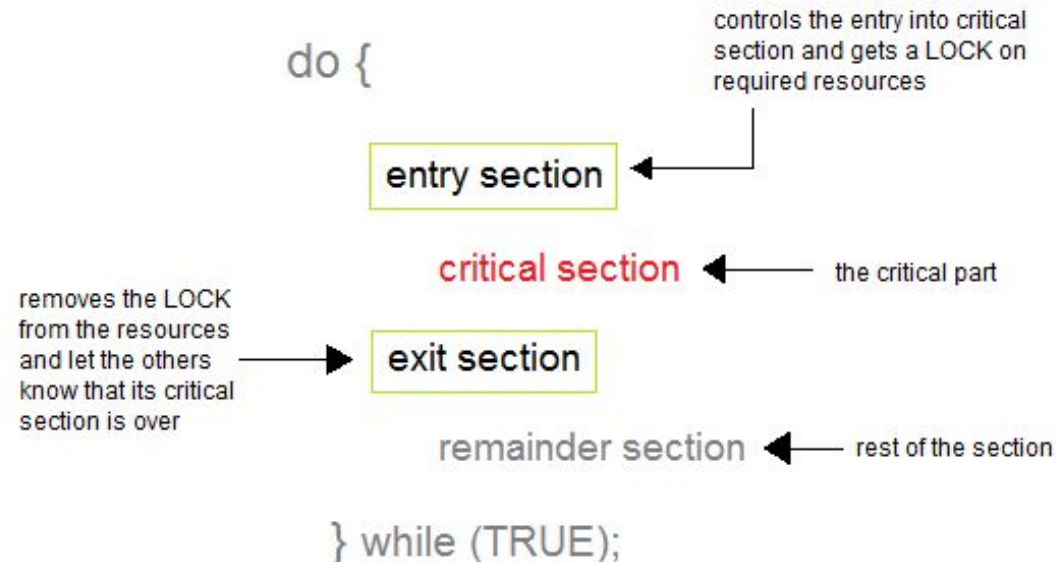




- Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

## Critical Section Problem

- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action.
- It means that in a group of cooperating processes, at a given point of time, only one process must be executing in its critical section.
- If any other process also wants to execute its critical section, it must wait until the first one finishes



# Rules for Critical Section Problem

All the Critical Section problems need to satisfy the following **three conditions**:

- **Mutual Exclusion:** If a process is in the critical section, then other processes shouldn't be allowed to enter into the critical section at that time i.e. only one process should execute in its critical section.
- **Progress:** If no process is executing in its critical section and some process wish to enter the critical section, then only those processes that are executing in its remainder section can participate.
- **Bounded Waiting:** There must be some limit to the number of times a process can go into the critical section i.e. there must be some upper bound. If no upper bound is there, then the same process will be allowed to go into the critical section again and again and other processes will never get a chance to get into the critical section.

# **Solutions To The Critical Section**

- In Process Synchronization, critical section plays the main role so that the problem must be solved.

## Synchronization Hardware

- Sometimes the problems of the Critical Section are also resolved by hardware.
- Some operating system offers a lock functionality where a Process acquires a lock when entering the Critical section and releases the lock after leaving it.
- So, when another process is trying to enter the critical section, it will not be able to enter as it is locked. It can only do so if it is free by acquiring the lock itself.

## Mutex Locks

- **Synchronization hardware** is not simple method to implement for everyone, so strict software method known as Mutex Locks was also introduced.
- In this approach, in the entry section of code, a **LOCK** is obtained over the critical resources used inside the critical section. In the exit section that lock is released.

# Semaphore Solution

- Semaphores are **integer variables** that are used to solve the critical section problem by using two **atomic operations, wait and signal** that are used for process synchronization.
- The definitions of wait and signal are as follows –

## Wait P(S), sleep, down

- The wait operation **decrements the value** of its argument S, if it is **positive**. If S is **negative or zero**, then **no operation is performed**.

```
wait(S)
{
    while (S<=0);
    S--;
}
```

## Signal V(S), wake-up, up

- The signal operation **increments the value of its argument S**.

```
signal(S)
{
    S++;
}
```

## Types of Semaphores


- There are two main types of semaphores i.e., counting semaphores and binary semaphores. Details about these are given as follows –

### Counting Semaphores:

- These are integer value semaphores and have an **unrestricted value domain**.
- These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.
- **If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.**

### Binary Semaphores:

- The binary semaphores are like counting semaphores, but their **value is restricted to 0 and 1**.
- The **wait operation** only works when the **semaphore is 1** and the **signal operation** succeeds when **semaphore is 0**.
- It is sometimes **easier** to implement binary semaphores than counting semaphores.



# Classical Problems of Synchronization

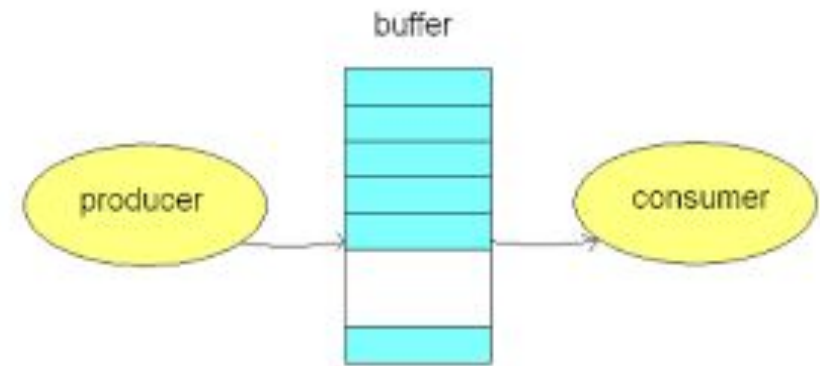
- Semaphore can be used in other synchronization problems besides Mutual Exclusion

## Three problems:

- Bounded Buffer (Producer-Consumer) Problem
- Race condition
- Dining Philosophers Problem
- The Readers Writers Problem

# Producer Consumer problem (Bounded Buffer)

- Producer Consumer problem is also called **Bounded Buffer problem**
- The Producer tries to insert data into empty slot of the buffer. A Consumer tries to remove data from filled slot in buffer.
- The problem describes two processes, the producer and the consumer, who share a common fixed size buffer.
- **Producer:** The producer's job is to generate a piece of data , put it into the buffer and start again.
- **Consumer:** The consumer is consuming the data (i.e removing it from the buffer) one piece at a time.
- If the buffer is empty, then the consumer should not try to access the data item from it.
- Similarly, a producer should not produce any data item if the buffer is full.
- **Counter:** counts the data item in a buffer or to track whether the buffer is empty or full. Counter is shared between two processes and updated by both.

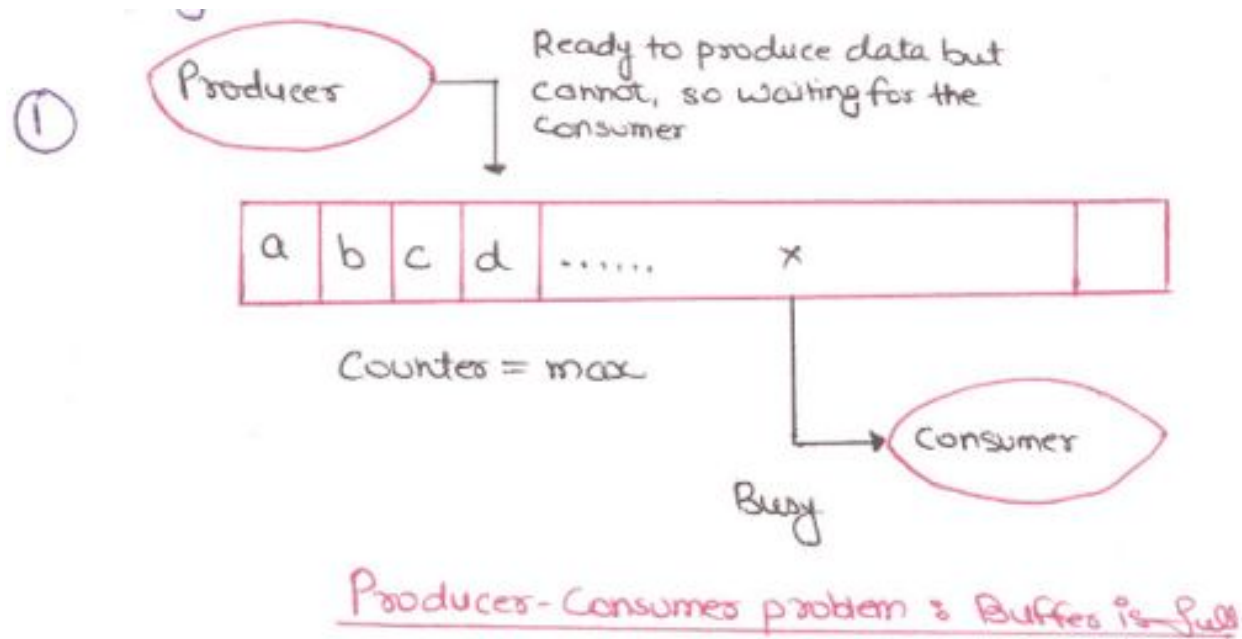




## How it works?

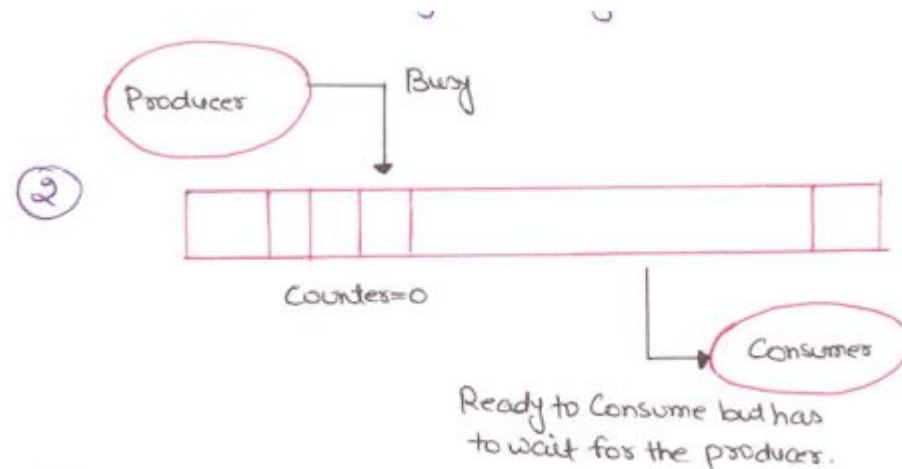
- Counter value is checked by Consumer before consuming it.
- If counter is 1 or greater than 1, then it starts executing the process and updates the counter.
- Similarly, Producer checks the buffer for the value of counter for adding data.
- If the counter is less than its maximum value, it means that there is some space in the buffer.
- It starts executing for producing the data item and updates the counter by incrementing it by one.

- Let  $\text{max}$  = maximum size of buffer
- If buffer is full then counter =  $\text{max}$  and consumer is busy executing other instructions or has not been allotted its time slice yet.



- In this buffer is full so producer has to wait until consumer sets counter by decrementing its value by 1.

- In this situation, buffer is empty, that is counter=0, and the producer is busy executing other instructions or has not been allotted its time slice yet. At this consumer is ready to consume an item from the buffer.
- Consumer waits until counter=1
- When the buffer is empty and producer busy in filling data items in buffer while consumer goes to sleep.
- When the counter goes to 1, then system generates WAKE UP calls to make consumer wakeup and start executing it.



# Race Condition

- It is the situation where several processes access and manipulates some data concurrently, and the outcome of execution depends on the order in which access takes place.
- When more than one processes are executing the same code or resource or any shared variable in that condition, there is a possibility that the output or the value of that shared variable is wrong, so for that all the process doing race to say that my output is correct, this condition is called as Race condition.
- In an Operating System, we have a number of processes and these processes require a number of resources. Now, think of a situation where we have two processes, and these processes are using the same variable "a". They are reading the variable and then updating the value of the variable and finally writing the data in the memory.

SomeProcess()

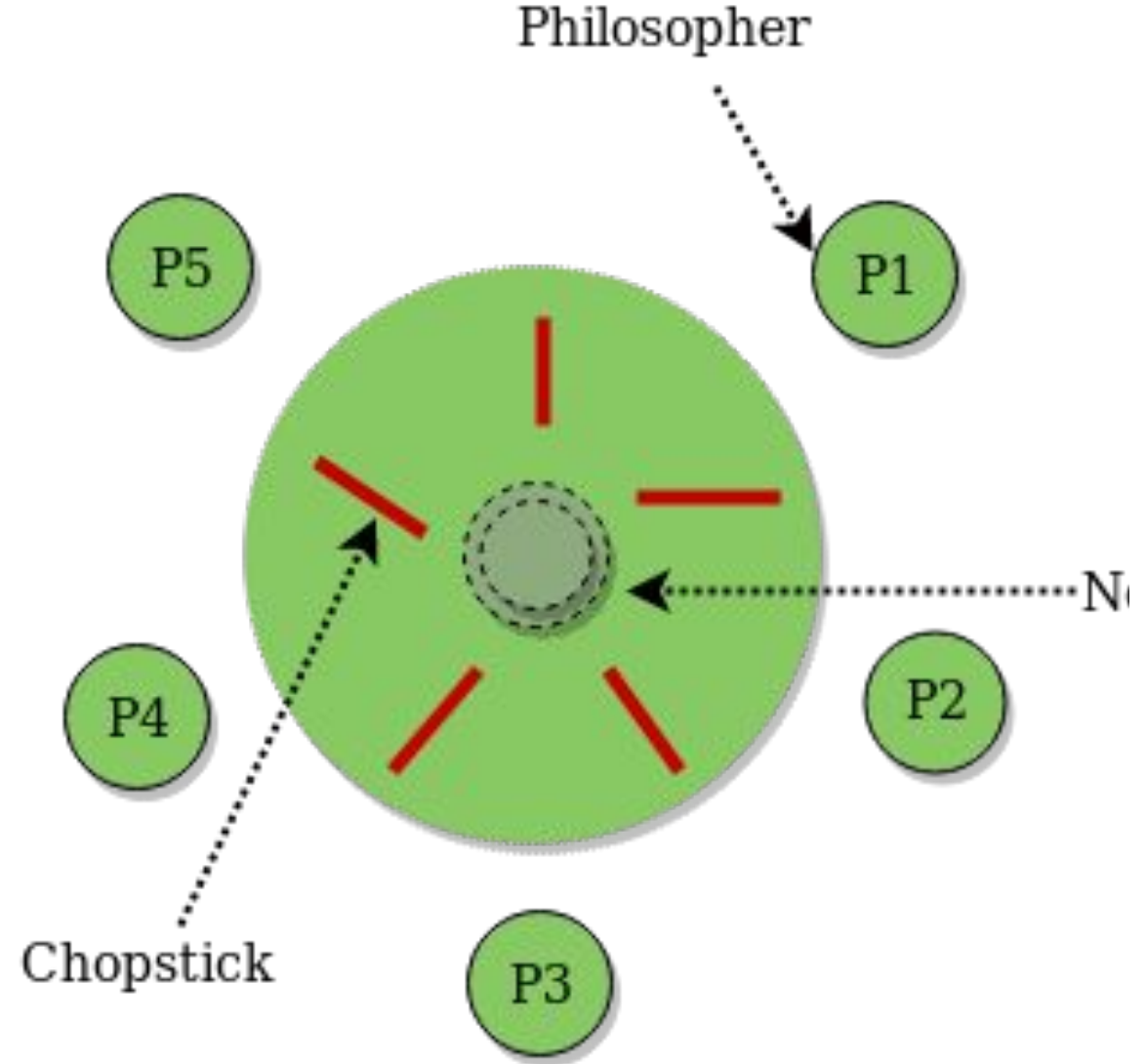
```
{  
• ...  
• read(a) //instruction 1  
• a = a + 5 //instruction 2  
• write(a) //instruction 3  
• ...  
• }
```

- In the above, you can see that a process after doing some operations will have to read the value of "a", then increment the value of "a" by 5 and at last write the value of "a" in the memory. Now, we have two processes P1 and P2 that needs to be executed.
- Let's take the following two cases and also assume that the value of "a" is 10 initially.
- In this case, process P1 will be executed fully (i.e. all the three instructions) and after that, the process P2 will be executed. So, the process P1 will first read the value of "a" to be 10 and then increment the value by 5 and make it to 15. Lastly, this value will be updated in the memory. So, the current value of "a" is 15. Now, the process P2 will read the value i.e. 15, increment with 5 ( $15+5 = 20$ ) and finally write it to the memory i.e. the new value of "a" is 20. Here, in this case, the final value of "a" is 20.
- In this case, let's assume that the process P1 starts executing. So, it reads the value of "a" from the memory and that value is 10(initial value of "a" is taken to be 10). Now, at this time, context switching happens between process P1 and P2.
- Now, P2 will be in the running state and P1 will be in the waiting state and the context of the P1 process will be saved. As the process P1 didn't change the value of "a", so, P2 will also read the value of "a" to be 10. It will then increment the value of "a" by 5 and make it to 15 and then save it to the memory. After the execution of the process P2, the process P1 will be resumed and the context of the P1 will be read. So, the process P1 is having the value of "a" as 10(because P1 has already executed the instruction 1). It will then increment the value of "a" by 5 and write the final value of "a" in the memory i.e.  $a = 15$ . Here, the final value of "a" is 15.

- In the above two cases, after the execution of the two processes P1 and P2, the final value of "a" is different i.e. in 1st case it is 20 and in 2nd case, it is 15. What's the reason behind this?
- The processes are using the same resource here i.e. the variable "a". In the first approach, the process P1 executes first and then the process P2 starts executing. But in the second case, the process P1 was stopped after executing one instruction and after that the process P2 starts executing. And here both the processes are dealing on the same resource i.e. variable "a" at the same time.
- Here, the order of execution of processes changes the output. All these processes are in a race to say that their output is correct. This is called a race condition.

# Dining Philosophers Problem

- Five philosophers are sitting around a circular table.
- Dining table has five chopsticks(fork) and bowl of rice in the middle.
- Philosopher can either eat or think.
- When a philosopher wants to eat (hungry), he uses two chopsticks.
- When a philosopher wants to think, he keeps down both chopsticks.
- Problem is “no philosopher will starve”.
- Every philosopher should eventually get a chance to eat.
- This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- A semaphore solution guarantees no two neighboring philosophers eat simultaneously but has the possibility of creating a deadlock.



### **Solution:**

- Philosopher must be allowed to pick up chopsticks if both left and right are available.
- Allow 4 philosopher to sit
- Philosopher tries to grab the chopstick by executing a **wait** operation.
- Philosopher releases her chopsticks by executing the **signal** operation.
- This solution guarantees that no two neighbors are eating simultaneously.
- So, now no deadlock condition.

### **For philosopher 1(P1):**

```
While(true)
{
Wait(chopstick[0]);
Wait(chopstick[1]);
//eat
Signal (chopstick[0]);
signal(chopstick[1]);
//think
}
```

### **For philosopher 2(P2):**

```
While(true)
{
Wait(chopstick[1]);
Wait(chopstick[2]);
//eat
Signal (chopstick[1]);
signal(chopstick[2]);
//think
}
```

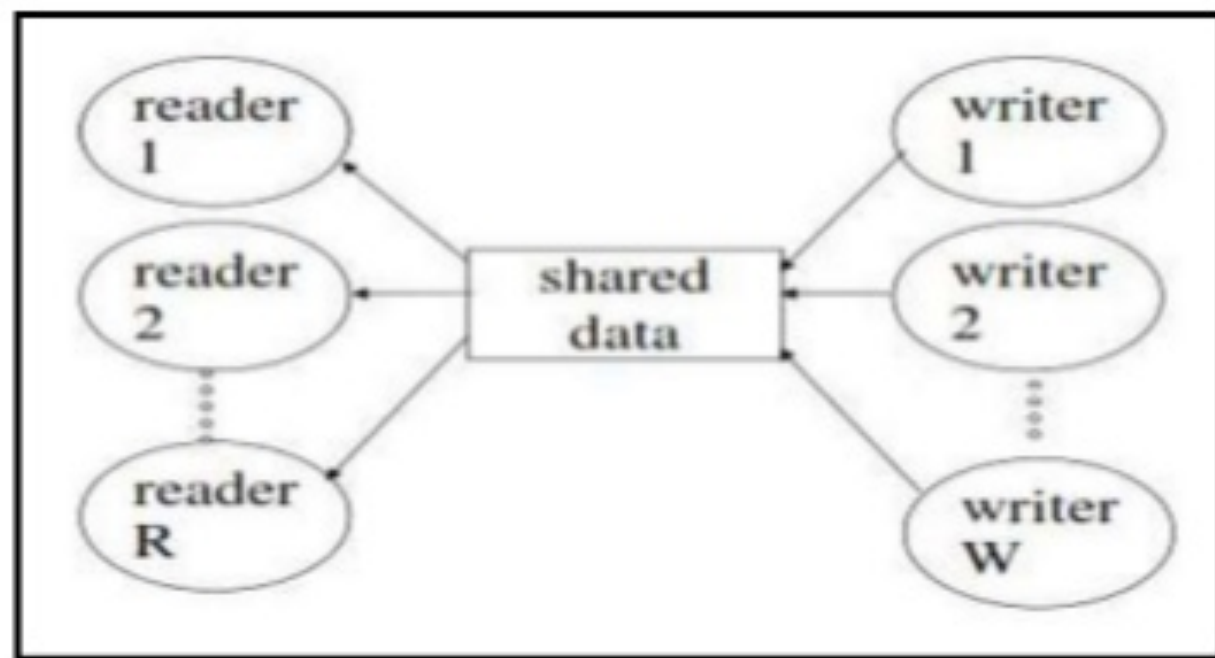


# The Readers Writers Problem

## Introduction to the problem:

- There is a shared resource which should be accessed by multiple processes.
- Multiple reader/writer activities can run simultaneously.
- At any time, a reader activity may wish to read data.
- Or a writer may wish to write or modify the data at any instant.
- Simultaneously, reading and writing to the data can cause inconsistency.
- Therefore, to avoid this situation, a set of rules is followed.
- Any number of readers can access the data(resource) simultaneously.
- But during the time a writer is writing, no other reader or writer may access the shared data.
- At the time of writing, no reading is allowed and at the time of reading, no writing is allowed.

# SHARING OF DATA



- **Readers:** only read the data set they do not perform any updates
- **Writers:** can both read and write



## **Semaphore solution**

### **Reader**

Wait(m); // acquire lock to access read\_count

read\_count++;

If(read\_count==1)

Wait(w) [first reader]

Signal (m); //releases lock

.

Wait(m);

Read\_count--;

If (read\_count ==0)

Signal (w); [last reader]

Signal (m);

}

### **Writer**

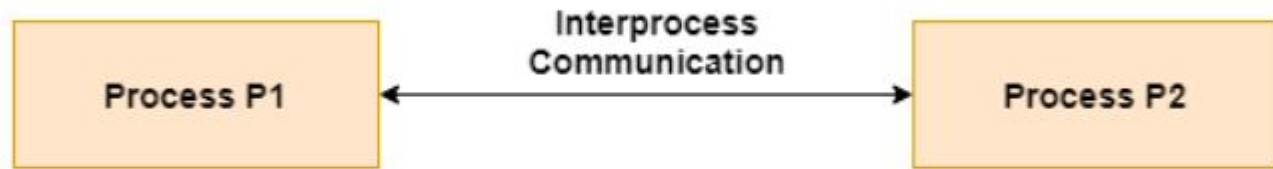
wait(w)

write;

Signal(w)

# Inter Process Communication in OS

- There are 2 types of process –
  - **Independent Processes** – Processes which do not share data with other processes .
  - **Cooperating Processes** – Processes that shares data with other processes.
- Cooperating process require Inter process communication (IPC) mechanism.
- Inter Process Communication is the mechanism by which cooperating process share data and information.
- There are 2 ways by which Inter process communication is achieved –
  - **Shared memory**
  - **Message Parsing**

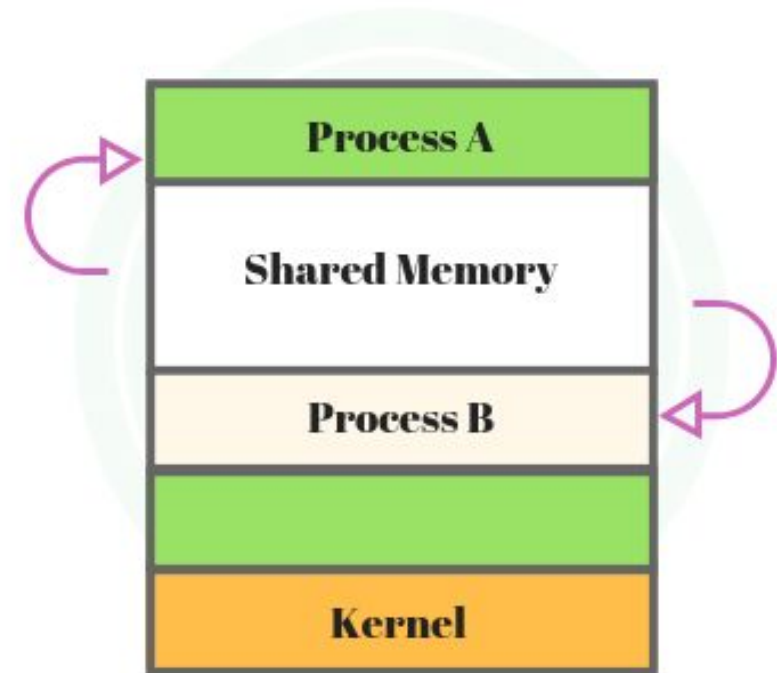


# Shared Memory

- A particular region of memory is shared between cooperating process.
- Cooperating process can exchange information by **reading and writing** data to this shared region.
- It's faster than Memory Passing, as Kernel is required only once, that is, setting up a shared memory . After That, kernel assistance is not required.
- **Ex: Producer-Consumer problem**
- There are two processes: Producer and Consumer.
- Producer produces some item and Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by Producer is stored and from which the Consumer consumes the item, if needed.



## Shared Memory in Operating System

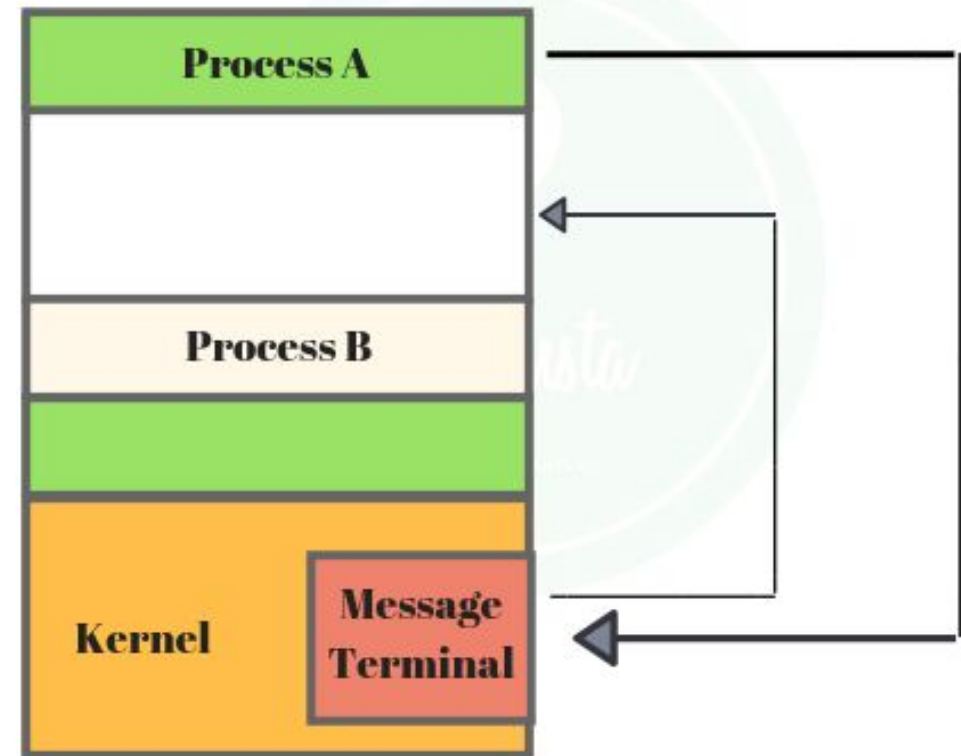


# Message Passing

- Communication takes place by exchanging messages directly between cooperating process.
- Easy to implement
- Useful for small amount of data.
- Implemented using System Calls, so takes more time than Shared Memory.



## Message Passing in Operating System



# Process Hierarchy

- Modern general-purpose operating system permits a user to create and destroy processes.
- A process may create several new processes during its time of execution. The creating process is called "Parent Process", while new processes are called "Child Processes".
- There are different possibilities concerning creating new processes:-
- **Execution:** The parent process continues to execute concurrently with its children processes or it waits until all of its children processes have terminated (sequentially).
- **Sharing:** Either the parent and children processes share all resources (like memory or files) or the children processes share only a subset of their parent's resources or the parent and children process share number of resources in common.

**A parent process can terminate equation of one of its children for one of these reasons:**

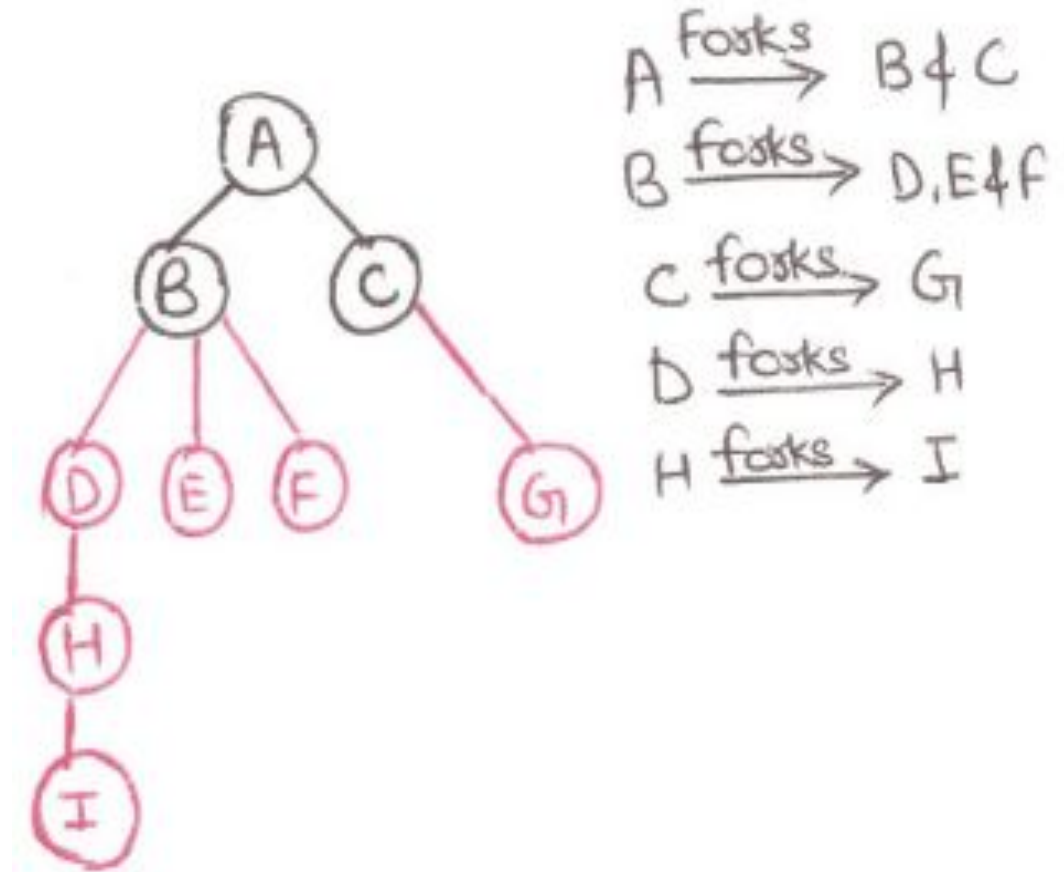
1) The child process has exceeded its usage of the resources it has been allocated. For this a mechanism must be available to allow the parent process inspect the state of its children process.

2) Task assigned to child process is no longer required.

- Let us discuss this using an example:
- In unix, this is done by the 'fork' system call , which creates a 'child' process and the 'exit' system call which terminates current process.
- The root of tree is a special process created by the OS during startup.

**A process can choose to wait for children to terminate**

- Example: C issued a wait() system call it would block until G finished.





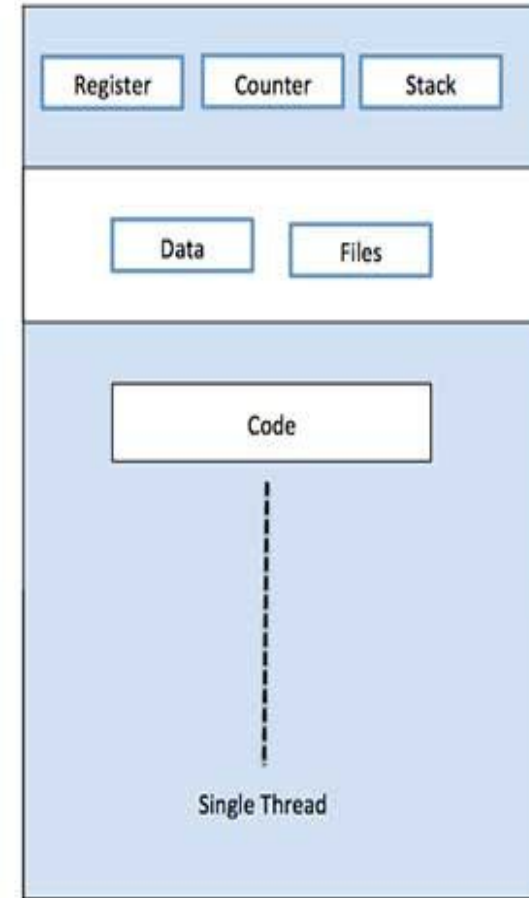
# Difference between Process and Thread

Parameter	Process	Thread
Definition	Process means a program is in execution.	Thread means a segment of a process.
Lightweight	The process is Heavyweight.	Threads are Lightweight.
Termination time	The process takes more time to terminate.	The thread takes less time to terminate.
System call	system call involved	No system call involved
Creation time	It takes more time for creation.	It takes less time for creation.
Communication	Communication between processes needs more time compared to thread.	Communication between threads requires less time compared to processes.
Context switching time	It takes more time for context switching.	It takes less time for context switching.
Resource	Process consume more resources.	Thread consume fewer resources.
Memory	The process is mostly isolated.	Threads share memory.
Shared	It does not share data.	It does share data with other threads.

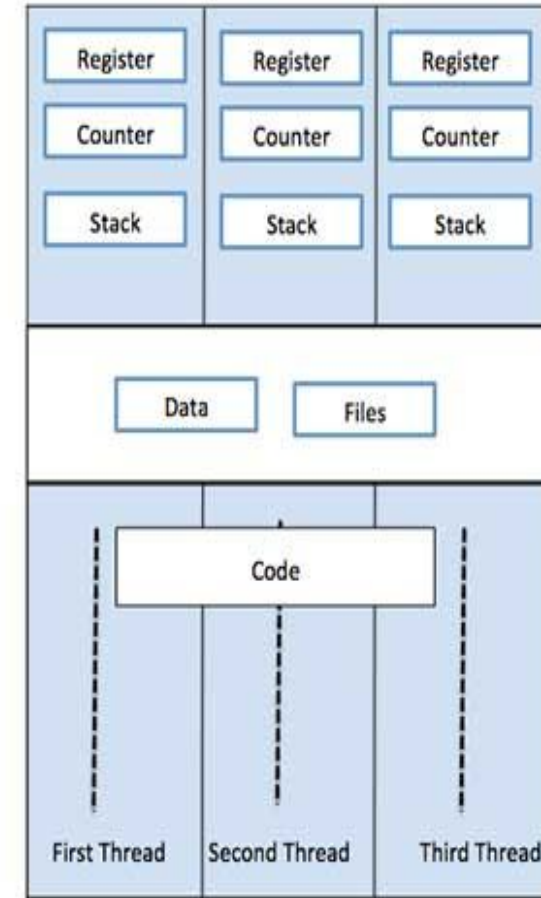
# Operating System - Multi-Threading

## What is Thread?

- A thread is a stream of execution throughout the process code having its program counter which keeps track of lists of instruction to execute next, system registers which bind its current working variables.
- Threads are also termed as **lightweight process**.
- A thread uses parallelism which provides a way to improve application performance.
- An operating system (OS) that has thread facility, the basic unit of CPU utilization is a thread.
- A thread is made up or consists of a program counter (PC), a register and a stack space.
- Threads are not independent of one another like processes as a result threads share with other threads their code section, data section, OS resources also referred to as task, such as open files and signals.
- Multithreading refers to multiple threads of execution within an operating system. In simple terms, two or more threads of a same process are executing simultaneously.
- The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

## Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

## Types of Thread

- Threads are implemented in following two ways –
- **User Level Threads** – User managed threads.
- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core.

# User Level Threads

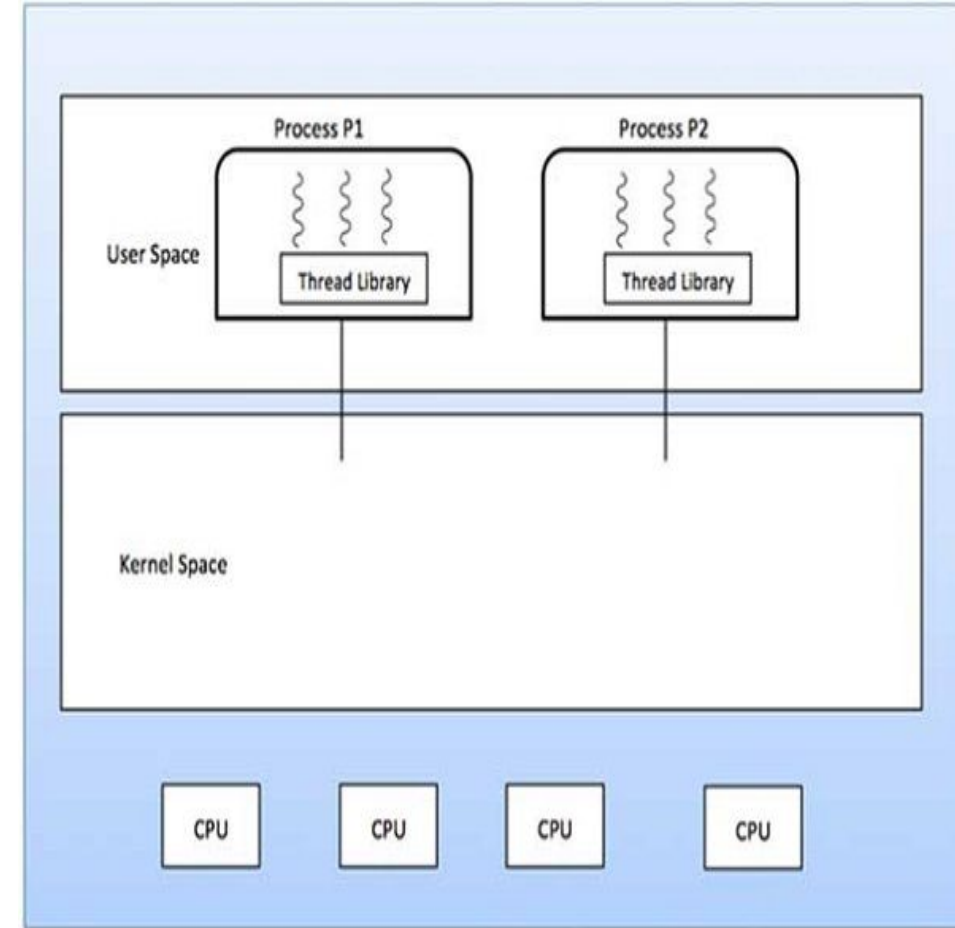
- User threads are implemented by users.
- Kernel knows nothing about user level threads.
- User level threads are loaded entirely in user space.
- Each process needs its own private thread table which consists of program counter, stack pointer, registers, state, etc.
- Scheduling are done in user space without the need of kernel intervention.

## Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- User level threads are fast to create and manage.
- User level threads allow each process to have its own scheduling algorithm.

## Disadvantages

- If one user level thread perform block system call, then entire process will be blocked.
- Multiprocessing cant be implemented using a single thread.



## Kernel Level Threads

- Kernel threads are implemented by OS.
- Thread management is done by the Kernel.
- No thread table in each process.
- Kernel has a thread table that keeps track of all the threads in a system.
- Information is stored in kernel instead of userspace.
- When a thread wants to create a new thread or destroy any existing thread, it makes a kernel call, which takes the action.
- The Kernel performs thread creation, scheduling and management in Kernel space.

## Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

## Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Context switching time is more, it requires hardware support.
- Implementation of kernel thread is complicated.

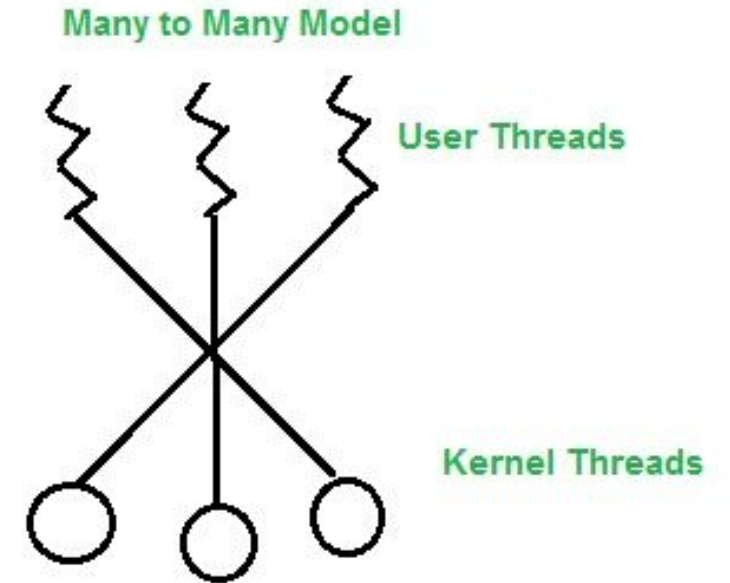
# Multithreading Models

- Some operating system provide a combined user level thread and Kernel level thread facility.
- **Solaris** is a good example of this combined approach.
- In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

Multithreading models are three types

## Many to Many Model

- Multiplexes many user-level threads to a smaller or equal no of kernel threads.
- Users can create any number of the threads.
- **Blocking the kernel system calls does not block the entire process.**
- Processes can be split across multiple processors.
- This model provides the **best accuracy on concurrency** and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

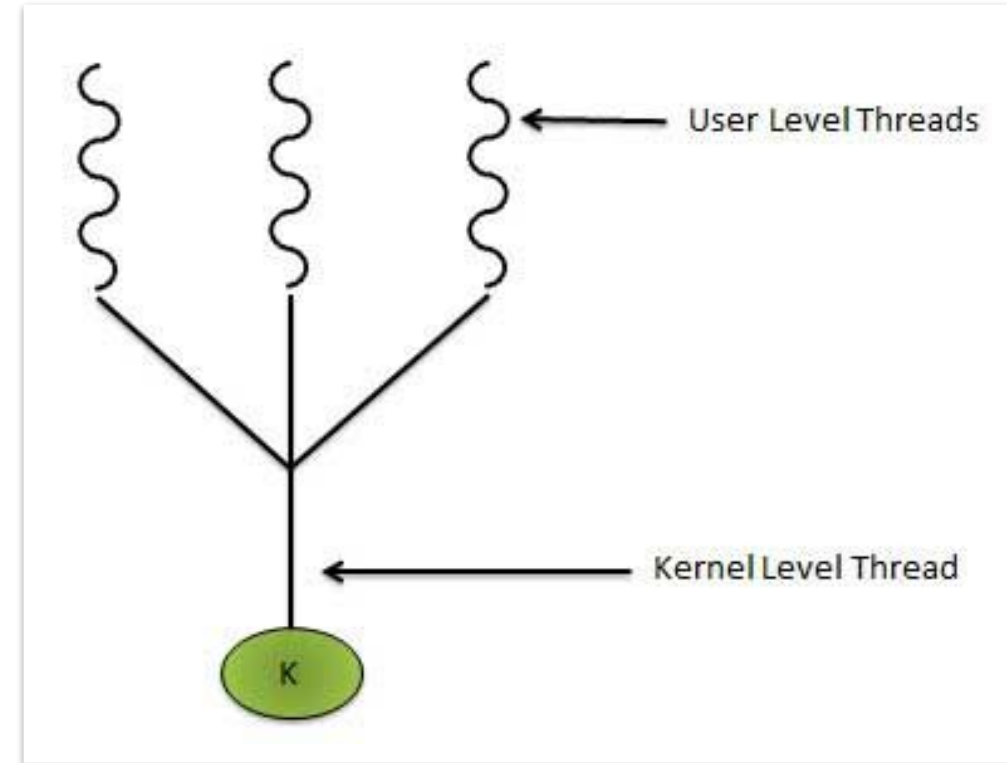


## Many to One Model

- Many-to-one model maps many user level threads to one Kernel-level thread.
- Thread management is done in user space by the thread library.

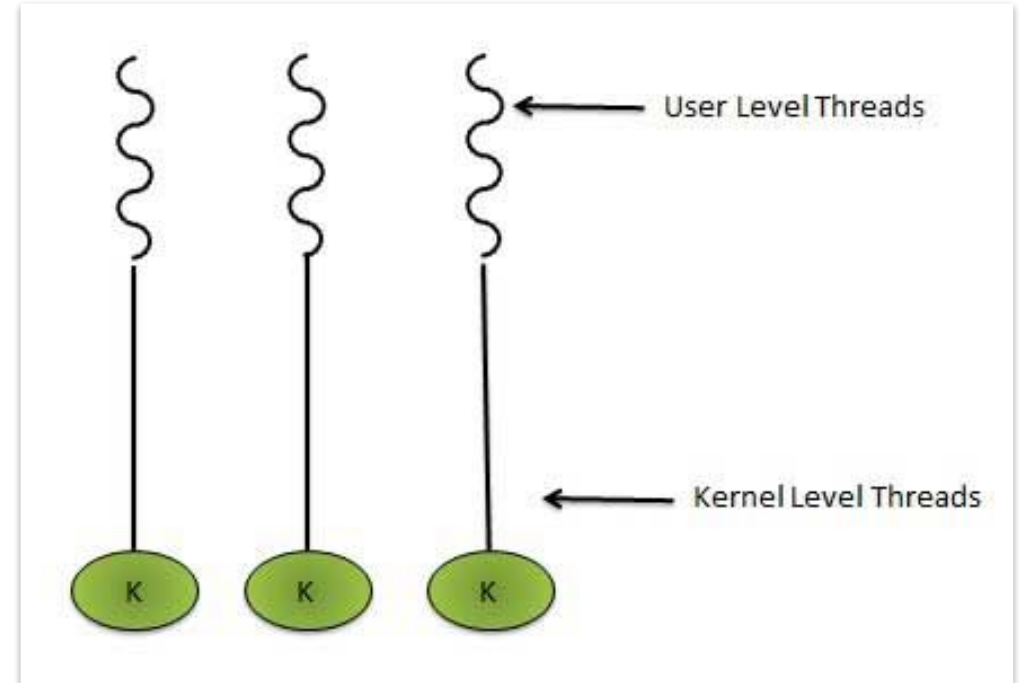
### Disadvantage:

- When thread makes a **blocking system call**, the **entire process will be blocked**.
- Only one thread can access the Kernel at a time, so multiple threads are **unable to run in parallel on multiprocessors**.



## One to One Model

- There is **one-to-one relationship** of user-level thread to the kernel-level thread.
- Maps each user level thread to a kernel level thread.
- This model provides **more concurrency** than the many-to-one model.
- It also **allows** another thread to run when a thread makes a **blocking system call**.
- It supports **multiple threads to execute in parallel** on microprocessors.
- **windows NT and windows 2000** use one to one relationship model.



### Disadvantage:

- creating user thread requires the corresponding Kernel thread.