

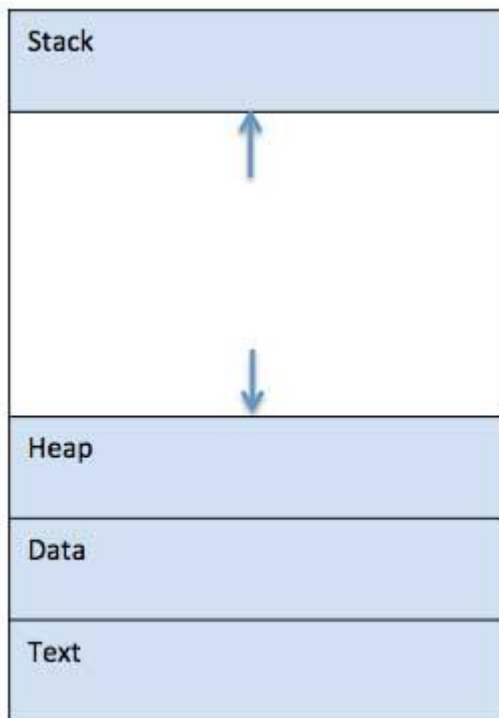
## 3. Process Management

### 3.1 Process

1. A process is basically a program in execution.
2. A process is defined as an entity which represents the basic unit of work to be implemented in the system.
3. Program becomes process when it loaded in memory

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

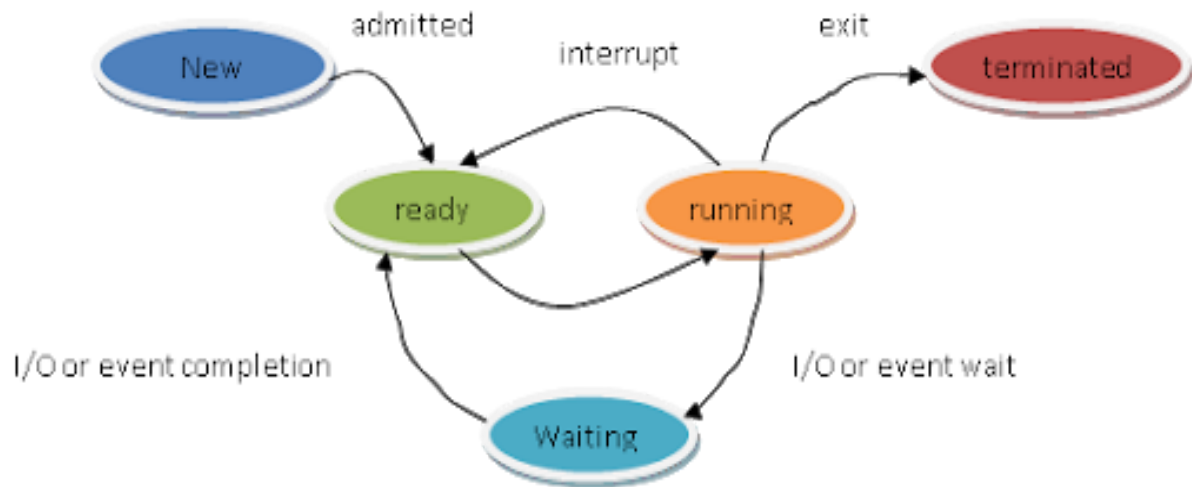
When a program is loaded into the memory and it becomes a process, it can be divided into four sections – stack, heap, text and data. The following image shows a simplified layout of a process inside main memory –



S.N	Component & Description
.	

1	<p>Stack</p> <p>The process Stack contains the temporary data such as method/function parameters, return address and local variables.</p>
2	<p>Heap</p> <p>This is dynamically allocated memory to a process during its run time.</p>
3	<p>Text</p> <p>This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.</p>
4	<p>Data</p> <p>This section contains the global and static variables.</p>

Process State :



**New:** The process is being created

**Running:** Instructions are being executed

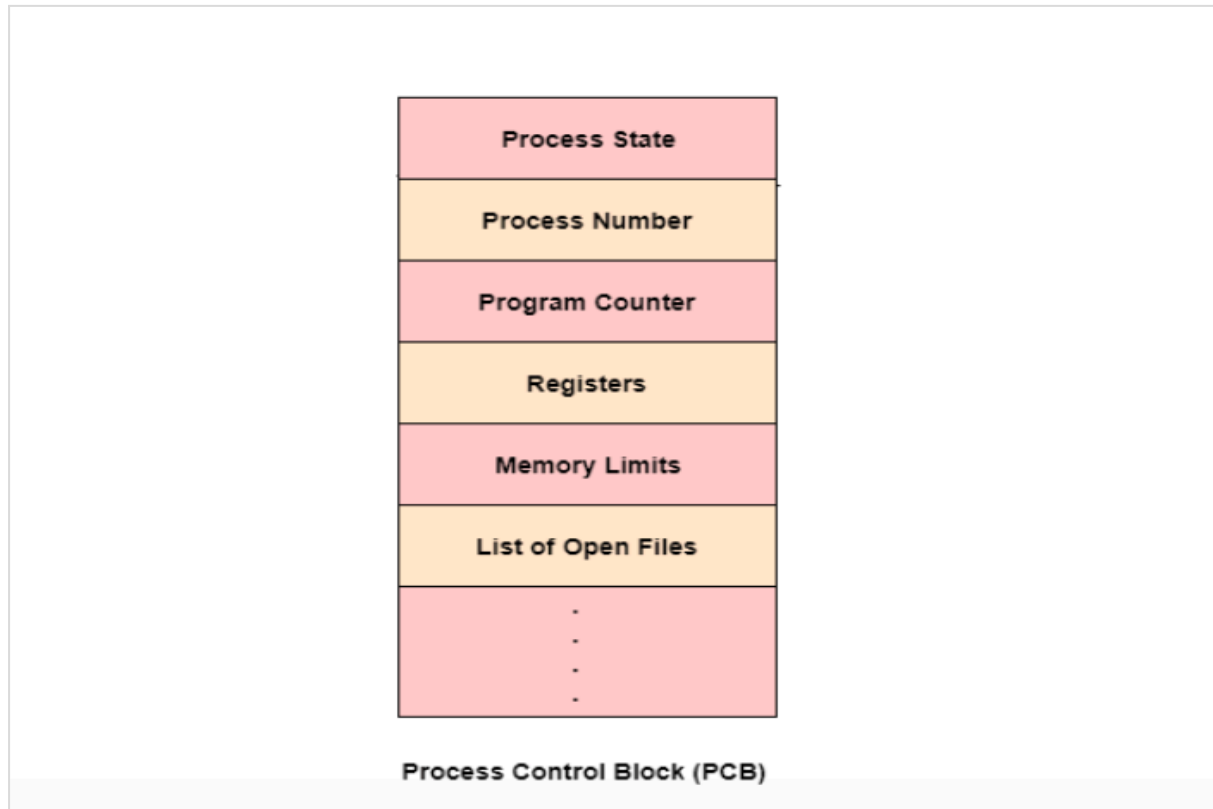
**Waiting:** The process is waiting for some event to occur.(I/O)

**Ready:** The process is waiting to be assigned to a processor

**Terminated:** The process has finished execution

## Structure of the Process Control Block

The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given diagram –



The following are the data items –

### Process State

This specifies the process state i.e. new, ready, running, waiting or terminated.

### Process Number

This shows the number of the particular process.

### Program Counter

This contains the address of the next instruction that needs to be executed in the process.

### Registers

This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

### List of Open Files

These are the different files that are associated with the process

### CPU Scheduling Information

The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

### **Memory Management Information**

The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

### **I/O Status Information**

This information includes the list of I/O devices used by the process, the list of files etc.

### **Accounting information**

The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

### **Location of the Process Control Block**

The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

## **3.2 Process Scheduling**

### **Definition**

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

#### **3.2.1 Process Scheduling Queues**

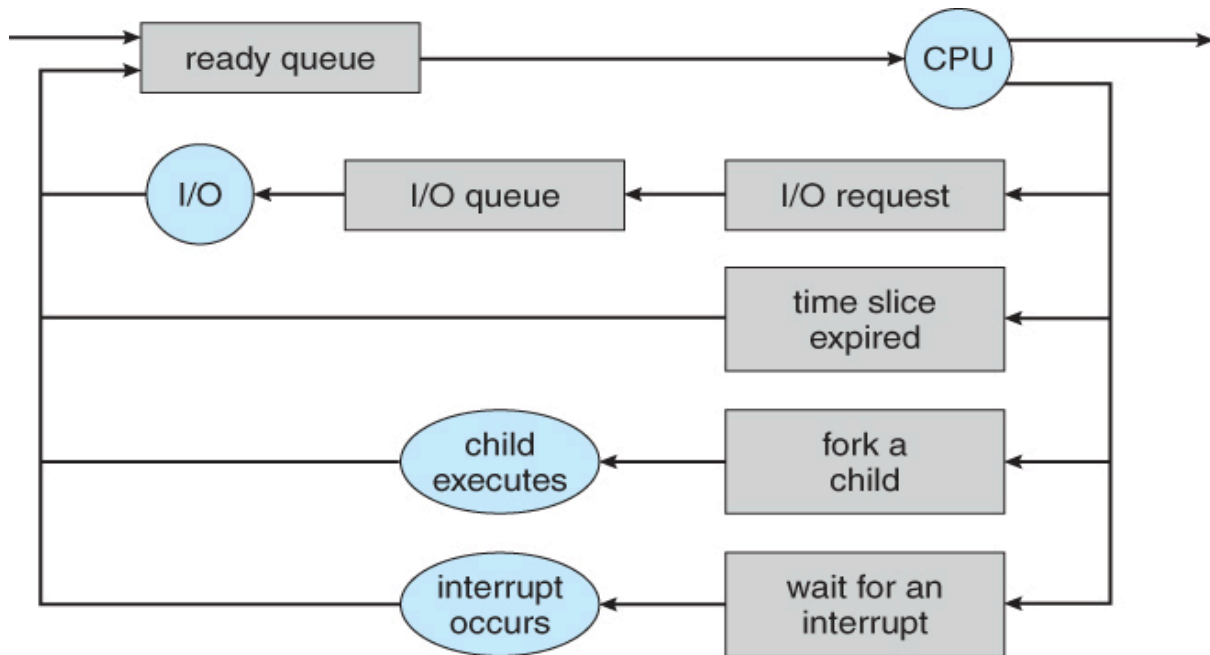
The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues

–

- **Job queue** – This queue keeps all the processes in the system.

- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.



**Queuing diagram representation of process scheduling**

### 3.2.2 Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types –

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

#### Long Term Scheduler

It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

## Short Term Scheduler

It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

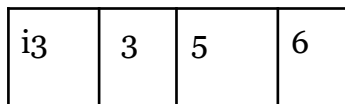
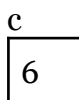
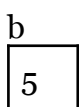
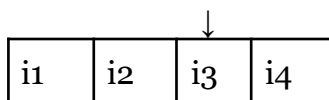
## Medium Term Scheduler

Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

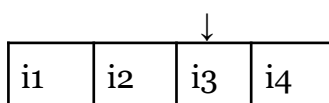
### 3.2.3 Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

**P1 -**



**P2 -**



a

12

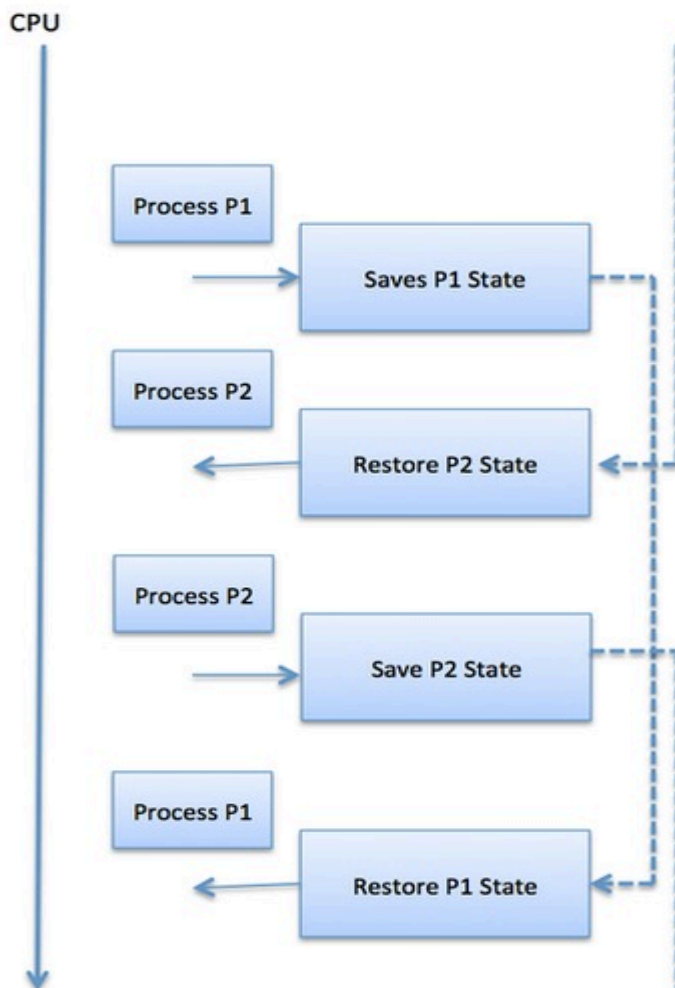
b

9

c

15

i3	12	9	15
----	----	---	----



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter



- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

### 3.3 Inter Process Communication:

Process can be of two types-

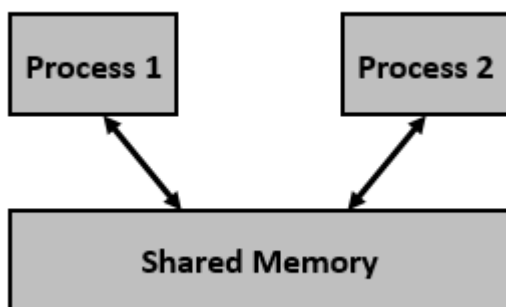
1. Independent Process

*Independent process* is the process that can not affect or be affected by the other processes. Independent processes do not share any data like temporary or persistent with any other process.

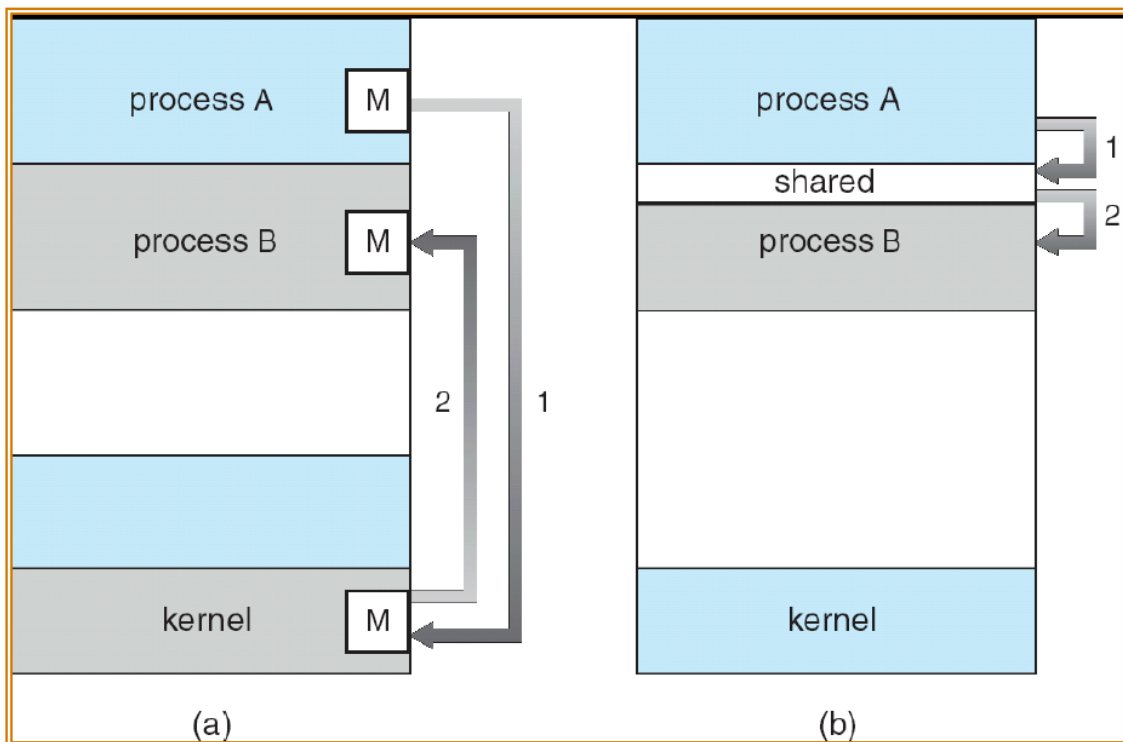
2. Co-operating process

*Cooperating processes* are affect or be affected by the other processes executing in the system. Cooperating process shares data with other processes.

Shared Memory :



Message Passing:

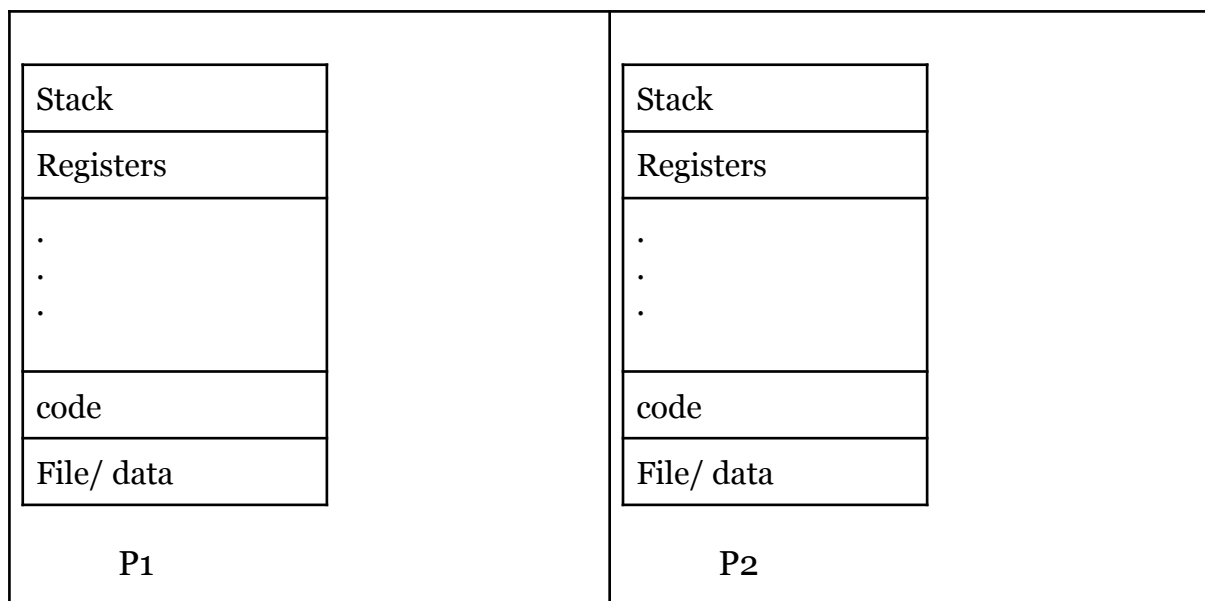


Message Passing

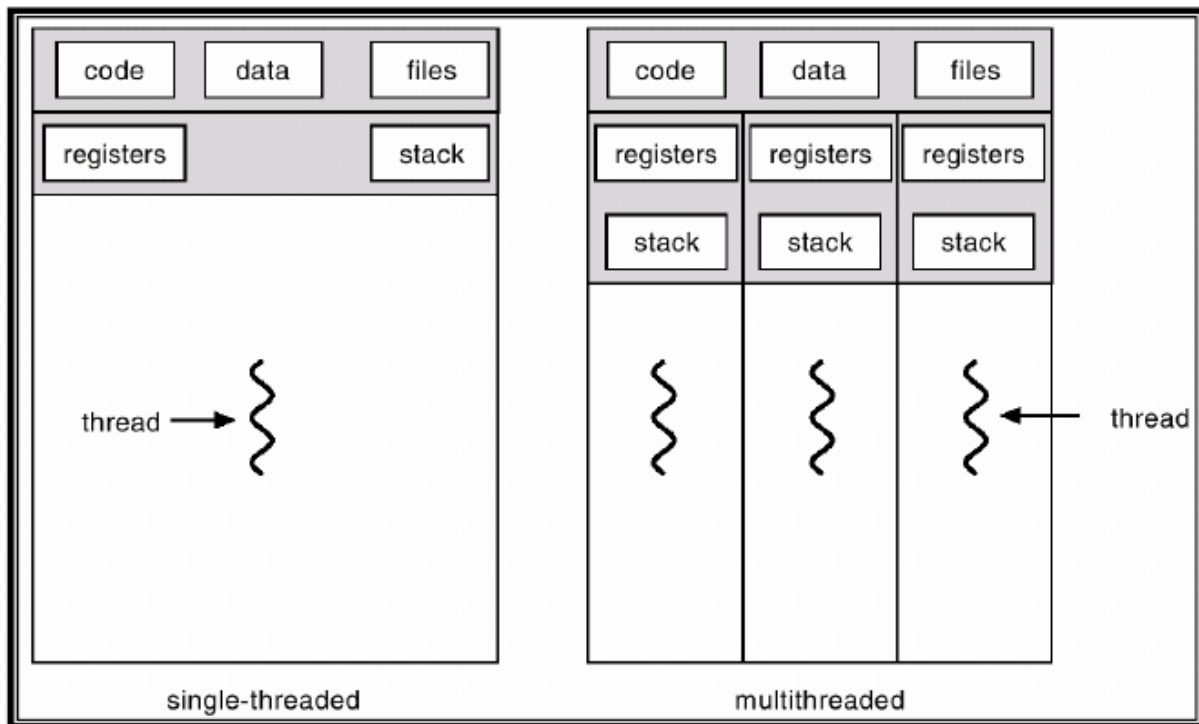
Shared Memory

✓ Observe: in a distributed system, message-passing is the only possible communication model.

Processes:



### 3.4 Threads:



Process	Thread
System calls are involved	No System calls are involved
All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Processes require more time for context switching .	Threads require less time for context switching.
Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.

Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .
If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.

## 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

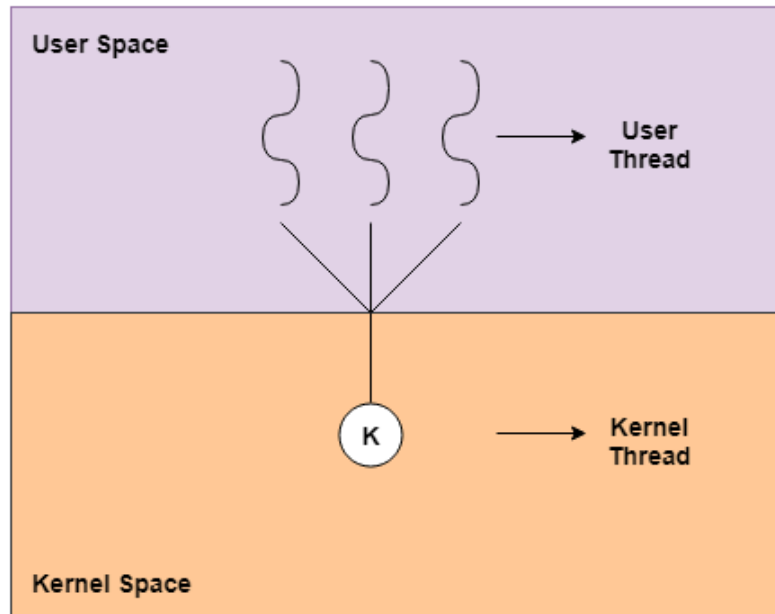
In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

### Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

### Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.



## Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

### 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.
- Kernel routines themselves can be multithreaded.

### Disadvantages

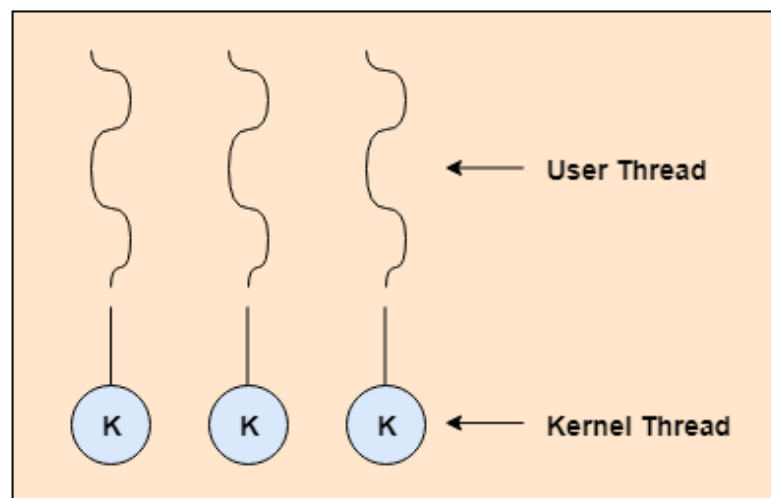
- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

User-level threads	Kernel-level threads
Managed by Users	Managed by operating System
No System calls are involved	System calls are involved
Context switching is faster	Context switching slower
User level threads are fast to create and manage.	Kernel threads are generally slower to create and manage than the user threads.

### 3.5 Multi-Threading Models

#### One to One Model:

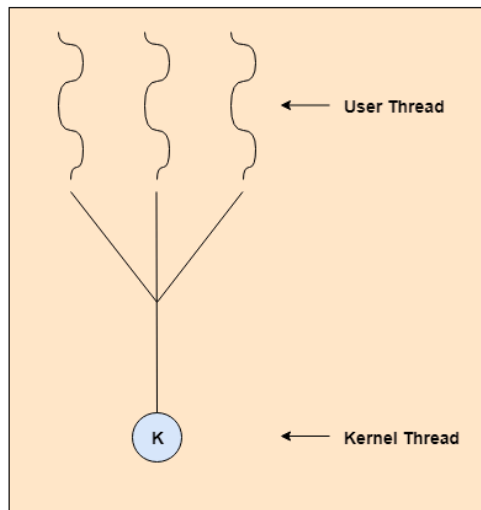
The one to one model maps each of the user threads to a kernel thread. This means that many threads can run in parallel on multiprocessors and other threads can run when one thread makes a blocking system call.



One to One Model

## Many to One Model

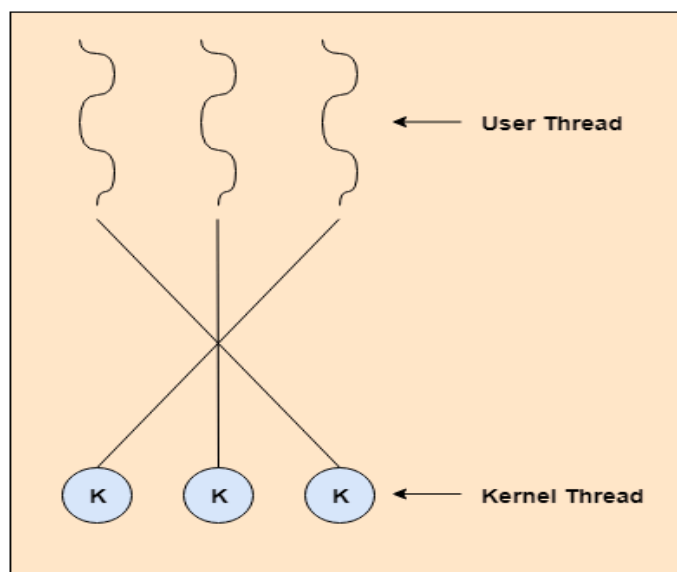
The many to one model maps many of the user threads to a single kernel thread. This model is quite efficient as the user space manages the thread management.



Many to One Model

## Many to Many Model

The many to many model maps many of the user threads to an equal number or lesser kernel threads. The number of kernel threads depends on the application or machine.



Many to Many Model

## 4. CPU Scheduling and Deadlock

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter –

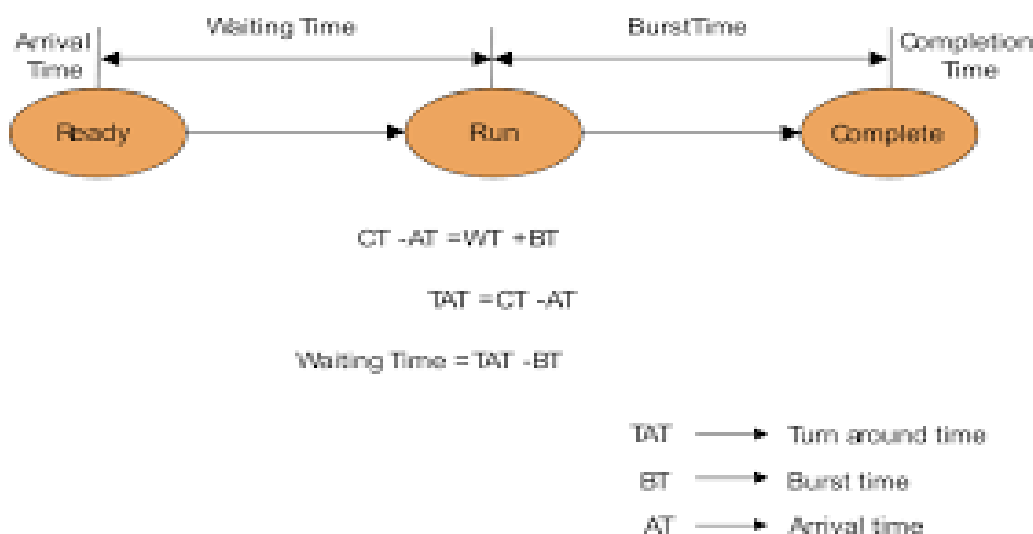
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

### Some Important Terms:

**1.CPU Scheduling:** CPU scheduling is a process that allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU.

The primary **objective** of **CPU scheduling** is to ensure that as many jobs are running at a time as is possible. On a single-CPU system, the goal is to keep one job running at all times. Multiprogramming allows us to keep many jobs ready to run at all times.



**2. Arrival Time:** The time at which process enters the ready queue.



**3. Burst Time:** The time required by a process to get executed on CPU. (Duration)

**4. Completion Time:** The time at which process completes its execution.

**5. Turnaround Time:** The total amount of time spent by the process from its arrival to its completion, is called Turnaround time.

(Completion time - Arrival Time)

**6. Waiting Time:** The Total amount of time for which the process waits for the CPU to be assigned is called waiting time.

(Turnaround Time - Burst Time)

**7. Response Time:** (the time at which process get CPU first time - Arrival Time)

## **4.2 Multilevel Queue Scheduling**

Multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

Multilevel queue scheduling has the following characteristics:

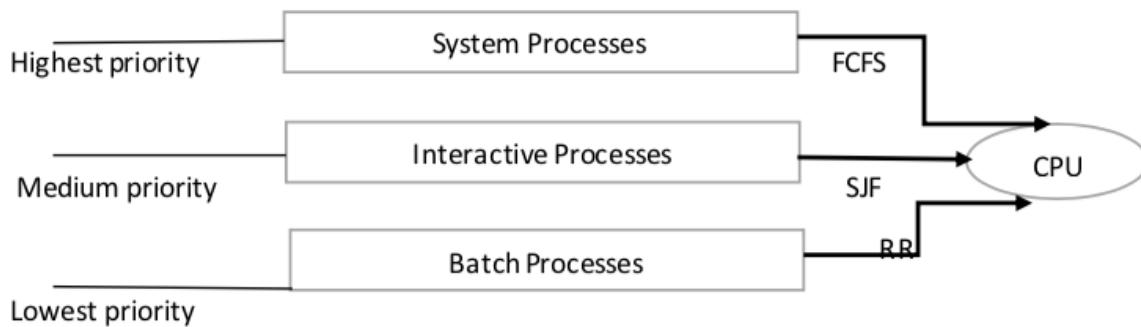
**Each queue has its own scheduling algorithm.**

For example given below, queue1(system process) uses FCFS (First Come First Serve), queue2(interactive process) uses SJF (Shortest Job First) while queue3 uses RR (Round Robin) to schedule their processes. For each queue, scheduling algorithm can be different or same.

**Processes are divided into different queues based on their process type, memory size and process priority.**

**Example of MultiLevel Queue Scheduling:**

Here, we will take 3 different types of processes called system processes, interactive processes and Batch processes. All the three processes have their own queue. Kindly look at below figure.



## MultiLevel Queue Scheduling Example

System process has the highest priority. If an interrupt is generated in the system, then the Operating system stops all the processes and handle the interrupt. According to different response time requirements, process needs different scheduling algorithm.

Interactive process has medium priority. If we are using VLC player, we directly interact with the application. So all these processes will come in an interactive process. These queues will use scheduling algorithm according to requirement.

Batch processes has the lowest priority. The processes which run automatically in the background comes under the batch processes. It is also known as background processes.

### 4.3 Deadlock:

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

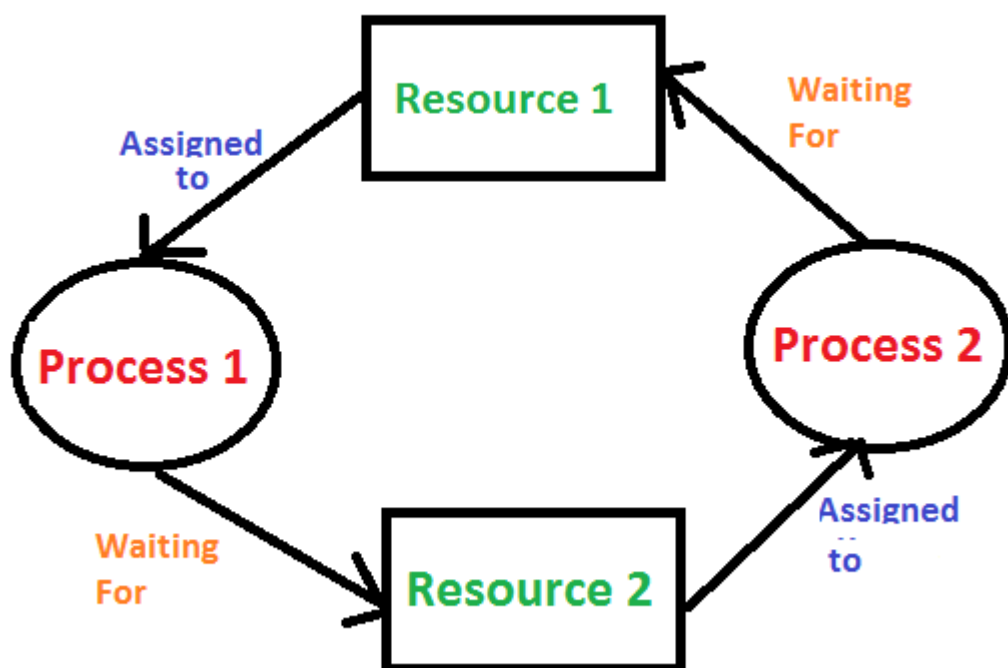
1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

In this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.



## **Necessary conditions for Deadlocks**

### **Mutual Exclusion**

A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

### **Hold and Wait**

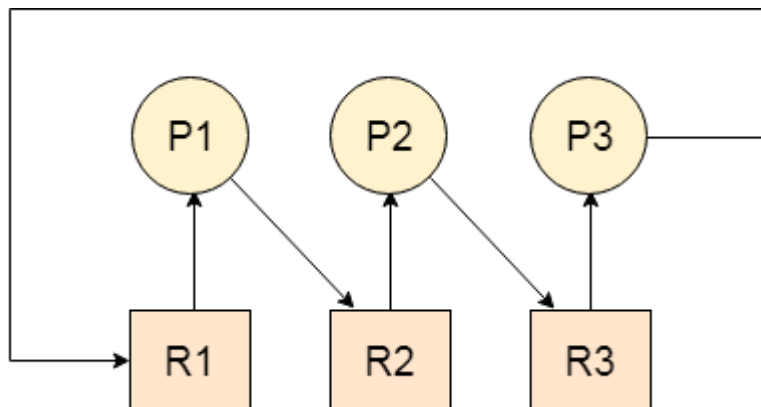
A process waits for some resources while holding another resource at the same time.

### **No preemption**

The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

### Circular Wait

All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.



## 4.4 Strategies for handling Deadlock

### 1. Deadlock Ignorance

Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.

There is always a tradeoff between Correctness and performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

### 2. Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.

We will discuss it later in detail.

### 3. Deadlock avoidance

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.

In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

We will discuss Deadlock avoidance later in detail.

### 4. Deadlock detection and recovery

This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

## 4.5 Deadlock Prevention

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

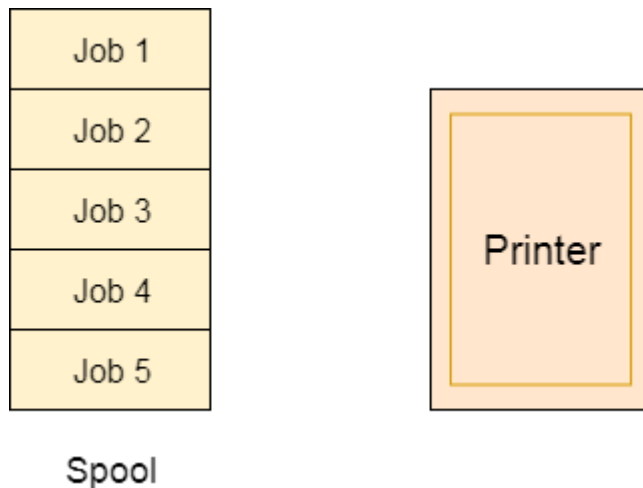
#### 1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

## Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

## 2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

**!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)**

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

### 3. No Preemption

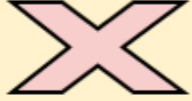



Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

### 4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	
Hold and Wait	Request for all the resources initially	
No Preemption	Snatch all the resources	
Circular Wait	Assign priority to each resources and order resources numerically	

Among all the methods, violating Circular wait is the only approach that can be implemented practically.

#### 4.6 Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

##### Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.

A state of a system recorded at some random time is shown below.



### Resources Assigned

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

### Resources still needed

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

1. **E** = (7 6 8 4)

2.  $P = (6 \ 2 \ 8 \ 3)$

3.  $A = (1 \ 4 \ 0 \ 1)$

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process.

Table 2 shows the instances of the resources, each process still needs. Vector E is the representation of total instances of each resource in the system.

Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use.

A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfill the request of all processes then the state of the system is called unsafe.

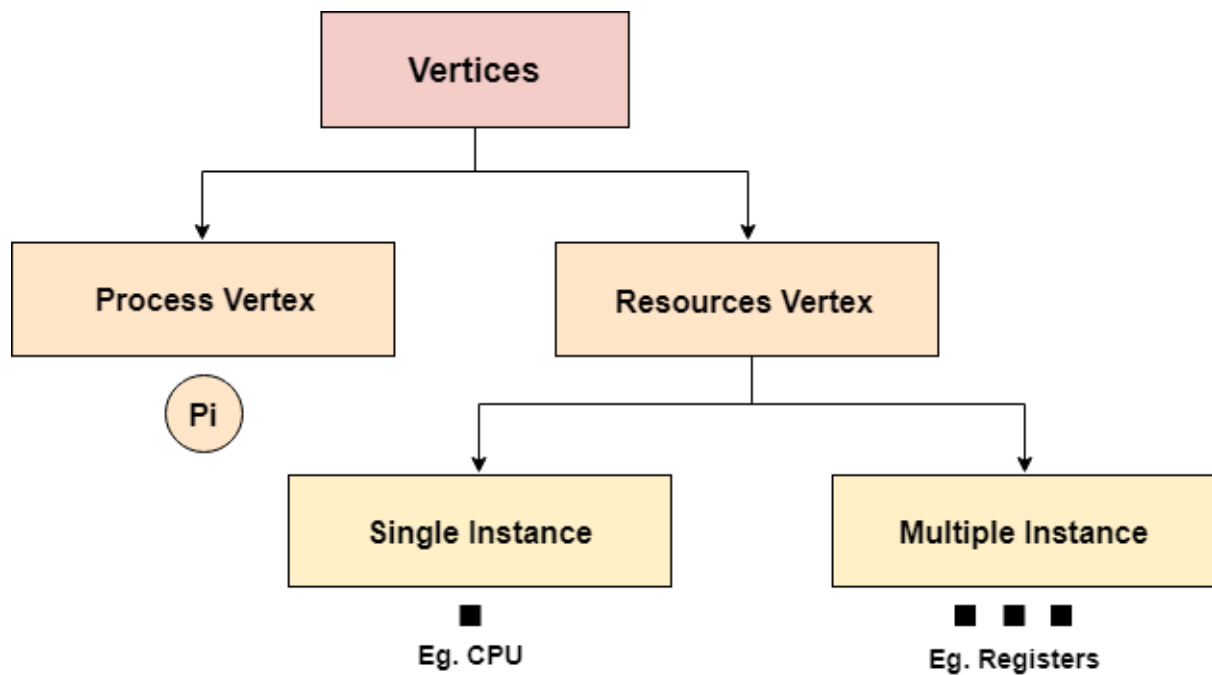
The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

## **Resource Allocation Graph**

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

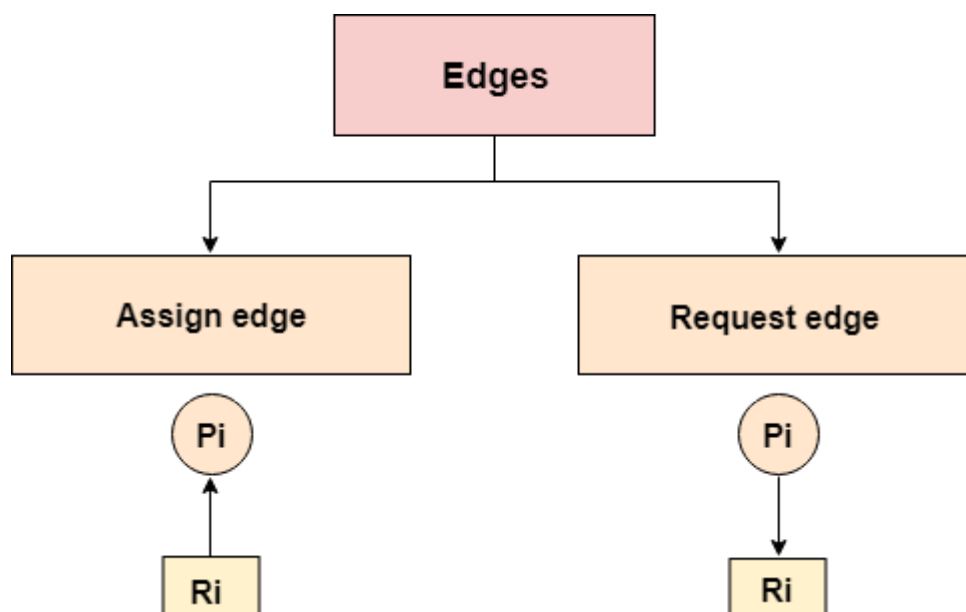
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.



Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource.

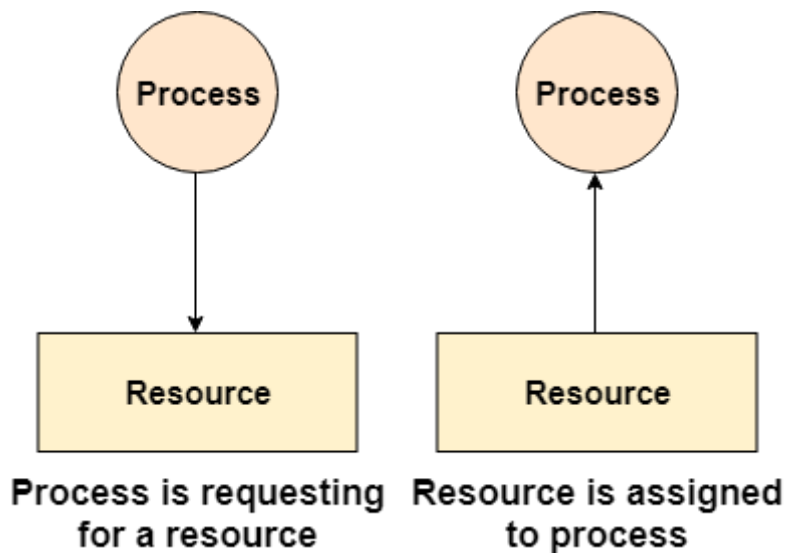
A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.

A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.

A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

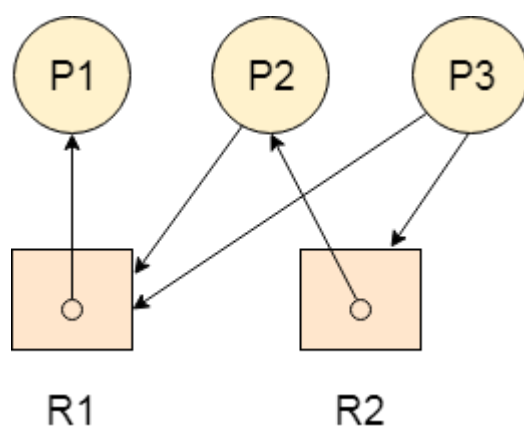


### Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.

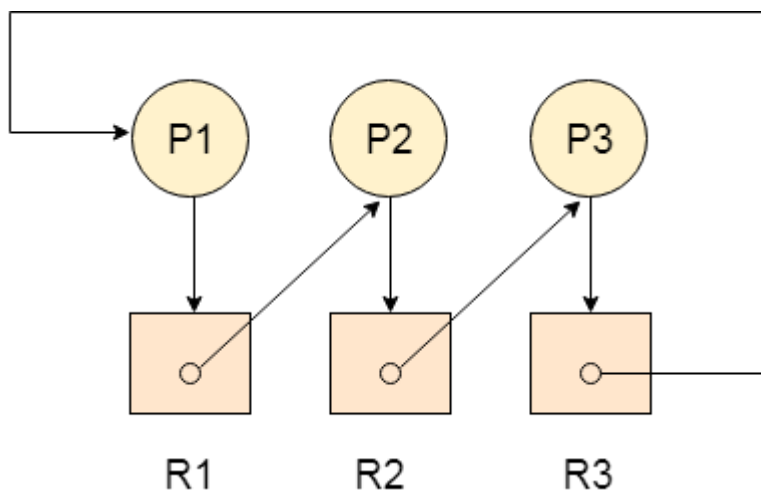


## Deadlock Detection using RAG

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

The following example contains three processes P1, P2, P3 and three resources R1, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

### Allocation Matrix

Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, an entry is being made in front of P1 and below R3 since R3 is assigned to P1.

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0

P3	0	1	0
----	---	---	---

### Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.

Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avail = (0,0,0)

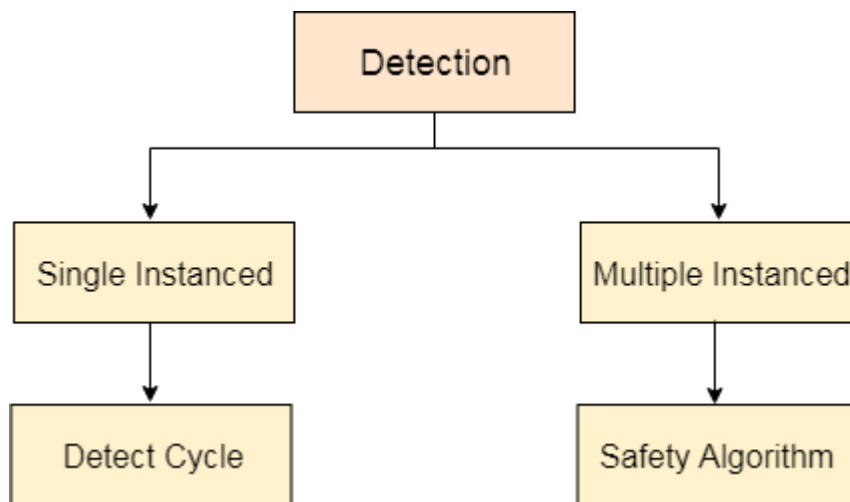
Neither we are having any resource available in the system nor a process going to release. Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.

We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

### Deadlock Detection and Recovery

In this approach, The OS doesn't apply any mechanism to avoid or prevent the deadlocks. Therefore the system considers that the deadlock will definitely occur. In order to get rid of deadlocks, The OS periodically checks the system for any deadlock. In case, it finds any of the deadlock then the OS will recover the system using some recovery techniques.

The main task of the OS is detecting the deadlocks. The OS can detect the deadlocks with the help of Resource allocation graph.



In single instanced resource types, if a cycle is being formed in the system then there will definitely be a deadlock. On the other hand, in multiple instanced resource type graph, detecting a cycle is not just enough. We have to apply the safety algorithm on the system by converting the resource allocation graph into the allocation matrix and request matrix.

In order to recover the system from deadlocks, either OS considers resources or processes.

### For Resource

#### Preempt the resource

We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.

#### Rollback to a safe state

System passes through various states to get into the deadlock state. The operating system can rollback the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state.

The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

### For Process

### Kill a process

Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, Operating system kills a process which has done least amount of work until now.

### Kill all process

This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.

