

Operating System

UNIT - III

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock.

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors.

Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.

Dead lock - System model

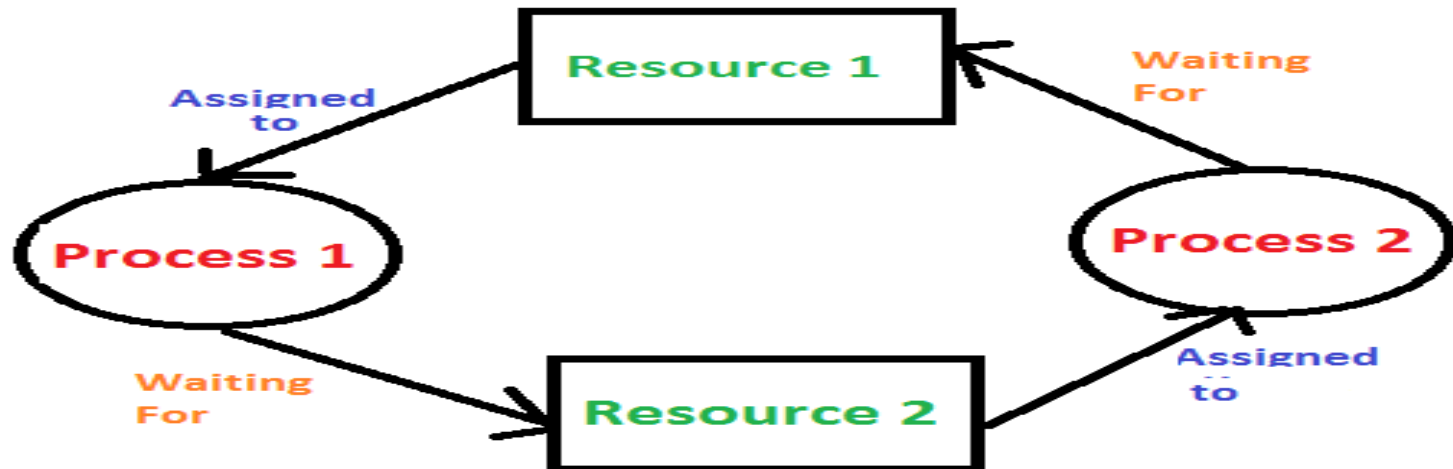
A process in operating systems uses different resources and uses resources in the following way.

- 1) Requests a resource
- 2) Use the resource
- 3) Releases the resource

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

A Situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s).

For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Deadlock Characterization

Deadlock can arise if the following 4 conditions hold simultaneously (Necessary Conditions) .

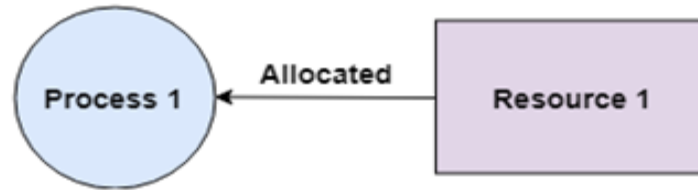
1. Mutual Exclusion

2. Hold and Wait

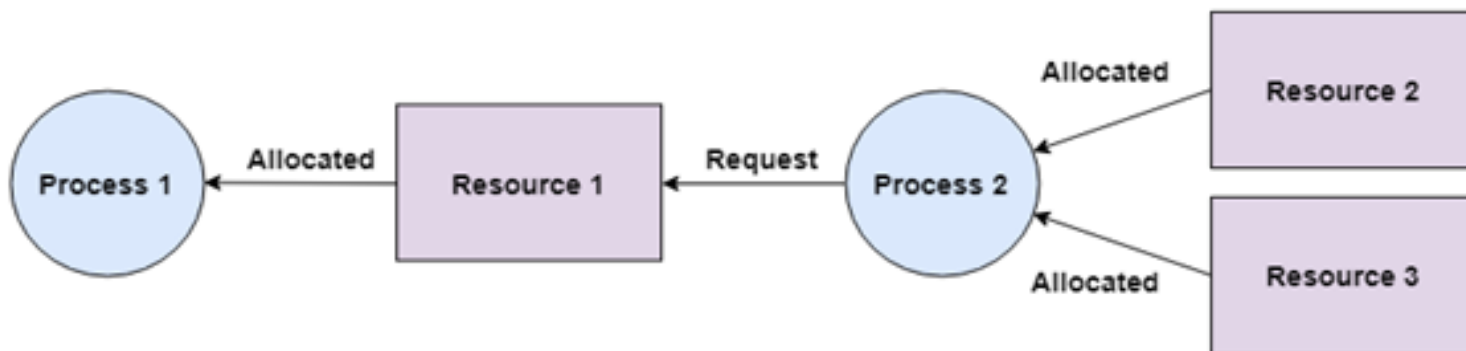
3. No Preemption

4. Circular Wait

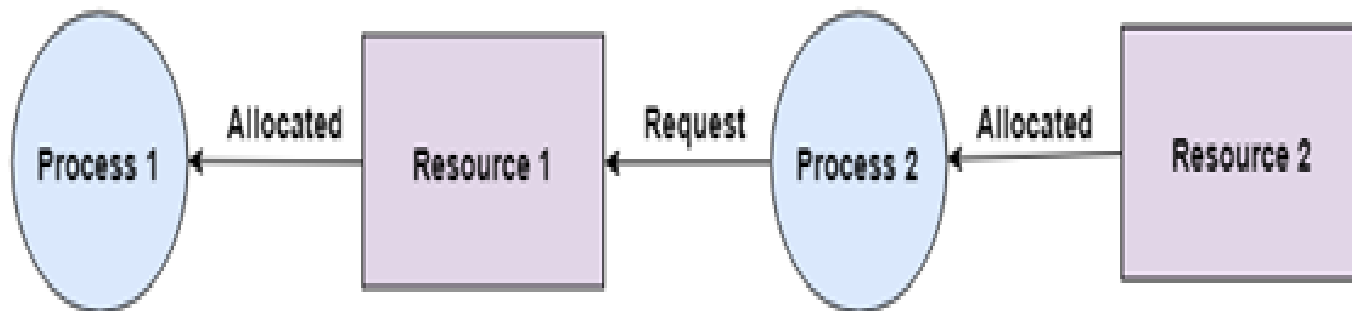
1. Mutual Exclusion: One or more than one resource are non-shareable (Only one process can use at a time).



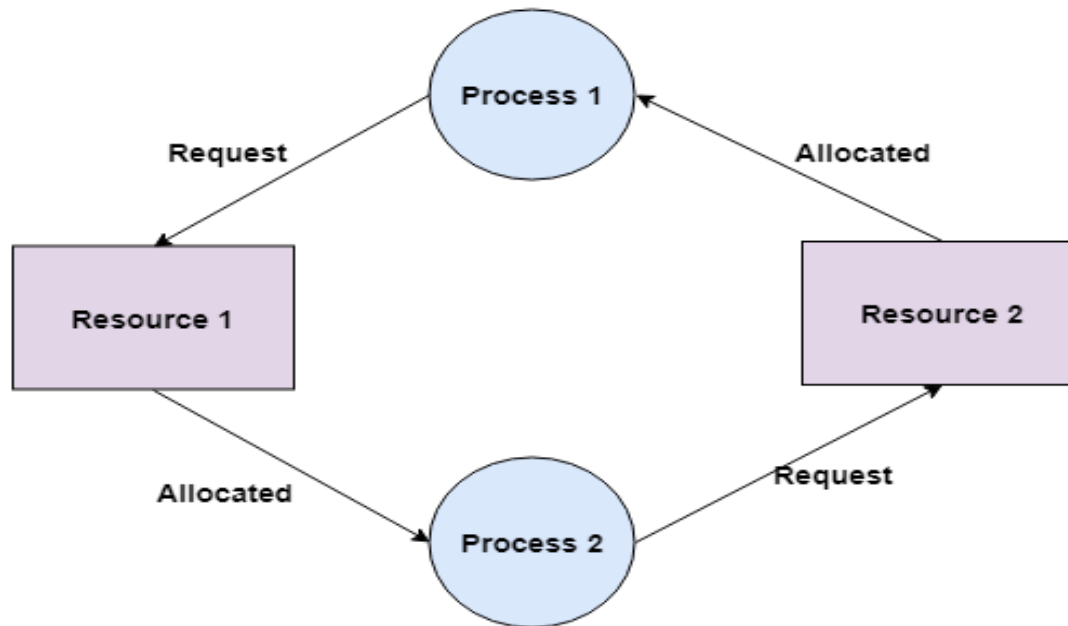
2. Hold and Wait: A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



3. No Preemption: A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4. **Circular Wait:** A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Methods for handling deadlock

There are three ways to handle deadlock

1) **Deadlock prevention or avoidance:** The idea is to not let the system into a deadlock state.

Prevention is done by negating one of above mentioned necessary conditions for deadlock.

Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use Banker’s algorithm in order to avoid deadlock.

2) **Deadlock detection and recovery:** Let deadlock occur, then do preemption to handle it once occurred.

3) **Ignore the problem altogether:** If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Methods of handling deadlocks :

There are three approaches to deal with deadlocks.

1. Deadlock Prevention
2. Deadlock avoidance
3. Deadlock detection

1. Deadlock Prevention :

The strategy of deadlock prevention is to design the system in such a way that the possibility of deadlock is excluded. Indirect method prevent the occurrence of one of three necessary condition of deadlock i.e., mutual exclusion, no pre-emption and hold and wait. Direct method prevent the occurrence of circular wait.

Prevention techniques –

Mutual exclusion – is supported by the OS.

Hold and Wait – condition can be prevented by requiring that a process requests all its required resources at one time and blocking the process until all of its requests can be granted at a same time simultaneously.

But this prevention does not yield good result because :

- long waiting time required
- in efficient use of allocated resource
- A process may not know all the required resources in advance.

No pre-emption – techniques for ‘no pre-emption are’

- If a process that is holding some resource, requests another resource that can not be immediately allocated to it, the all resource currently being held are released and if necessary, request them again together with the additional resource.
- If a process requests a resource that is currently held by another process, the OS may pre-empt the second process and require it to release its resources. This works only if both the processes do not have same priority.

Circular wait One way to ensure that this condition never hold is to impose a total ordering of all resource types and to require that each process requests resource in an increasing order of enumeration, i.e., if a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in ordering.

2. Deadlock Avoidance :

This approach allows the three necessary conditions of deadlock but makes judicious choices to assure that deadlock point is never reached. It allows more concurrency than avoidance detection

A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to deadlock. It requires the knowledge of future process requests.

Two techniques to avoid deadlock :

- Process initiation denial
- Resource allocation denial

Advantages of deadlock avoidance techniques :

- Not necessary to pre-empt and rollback processes
- Less restrictive than deadlock prevention

Disadvantages :

- Future resource requirements must be known in advance
- Processes can be blocked for long periods
- Exists fixed number of resources for allocation

3. Deadlock Detection :

Deadlock detection is used by employing an algorithm that tracks the circular waiting and killing one or more processes so that deadlock is removed. The system state is examined periodically to determine if a set of processes is deadlocked. A deadlock is resolved by aborting and restarting a process, relinquishing all the resources that the process held.

- This technique does not limit resources access or restrict process action.
- Requested resources are granted to processes whenever possible.
- It never delays the process initiation and facilitates online handling.
- The disadvantage is the inherent pre-emption losses.

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.

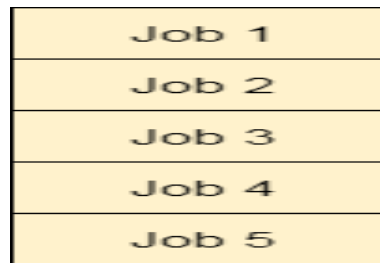
1. Mutual Exclusion

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.



Spool



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

- This cannot be applied to every resource.

After some point of time, there may arise a **race condition** between the processes to get space in that pool.

- 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:

- Practically not possible.
- 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.

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

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	

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.

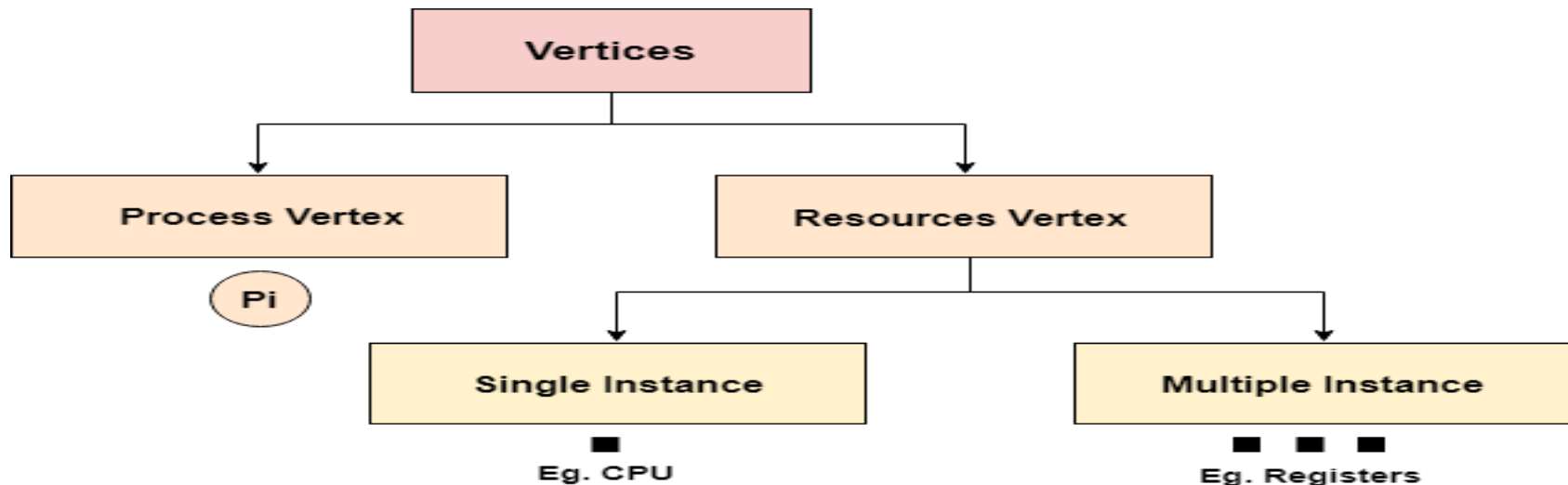
Resource Allocation Graph

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.

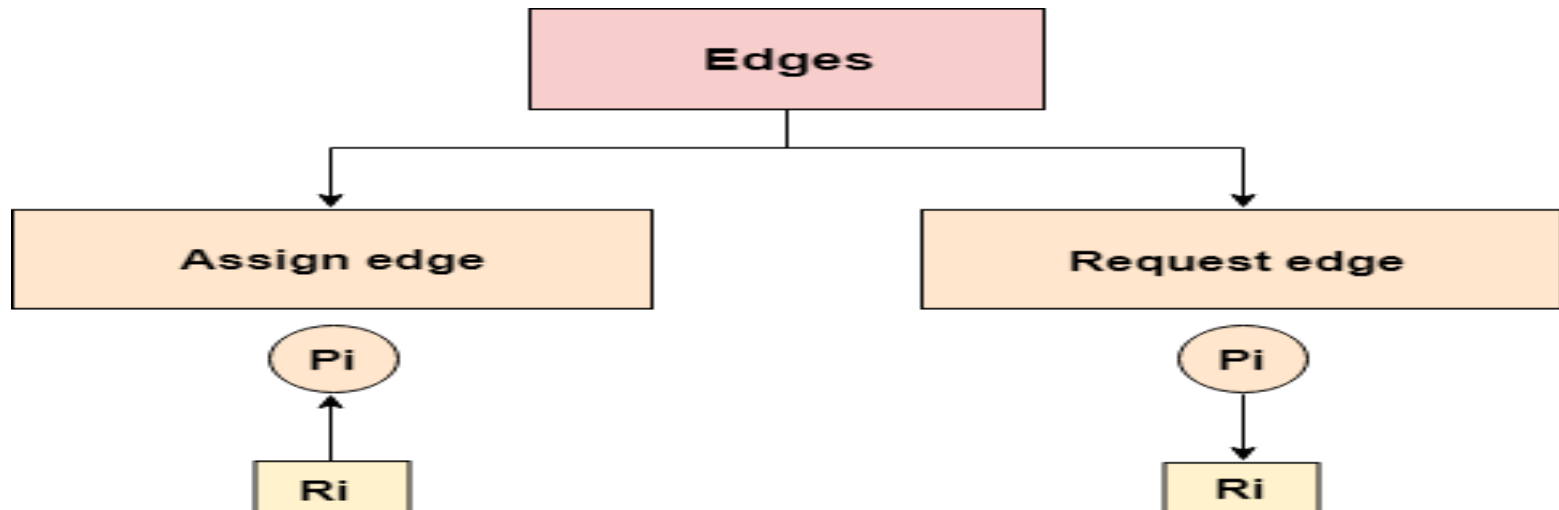
In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.

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.

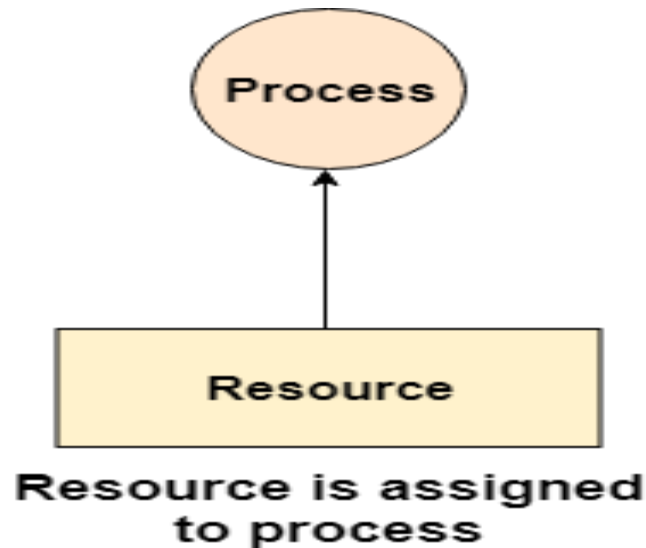
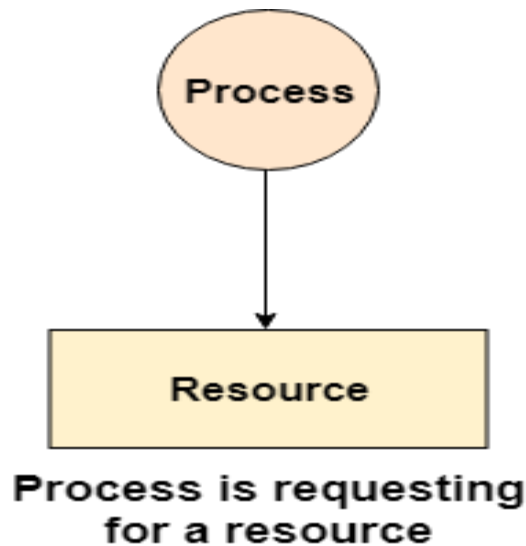


Edges in RAG(Resource Allocation Graph) are also of two types, one represents assignment and other represents the wait of a process for a resource.

The above image shows each of them.

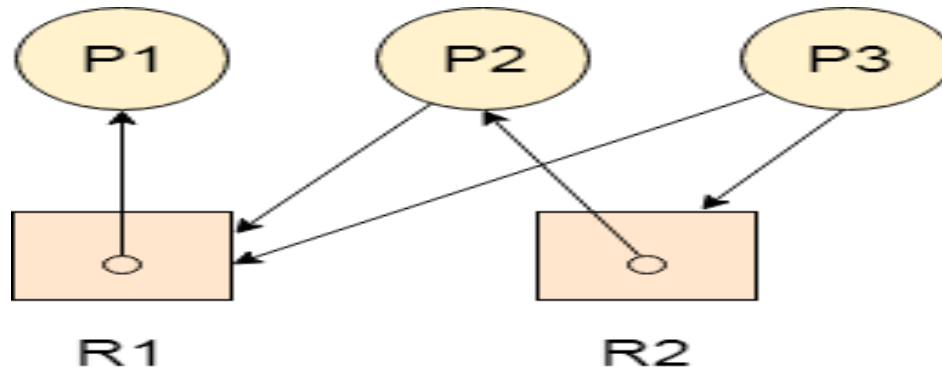
A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.

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.



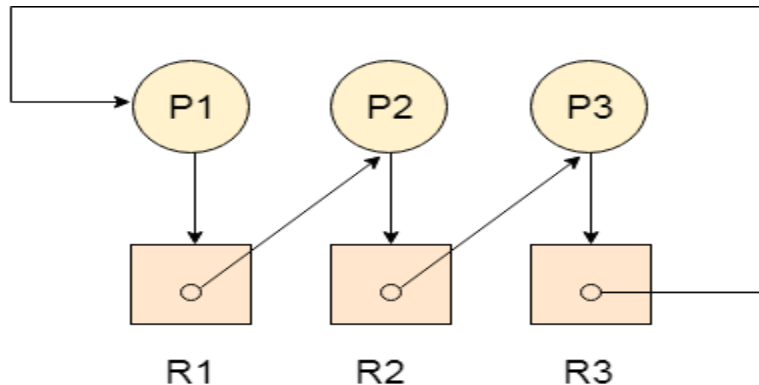
Example

- 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.

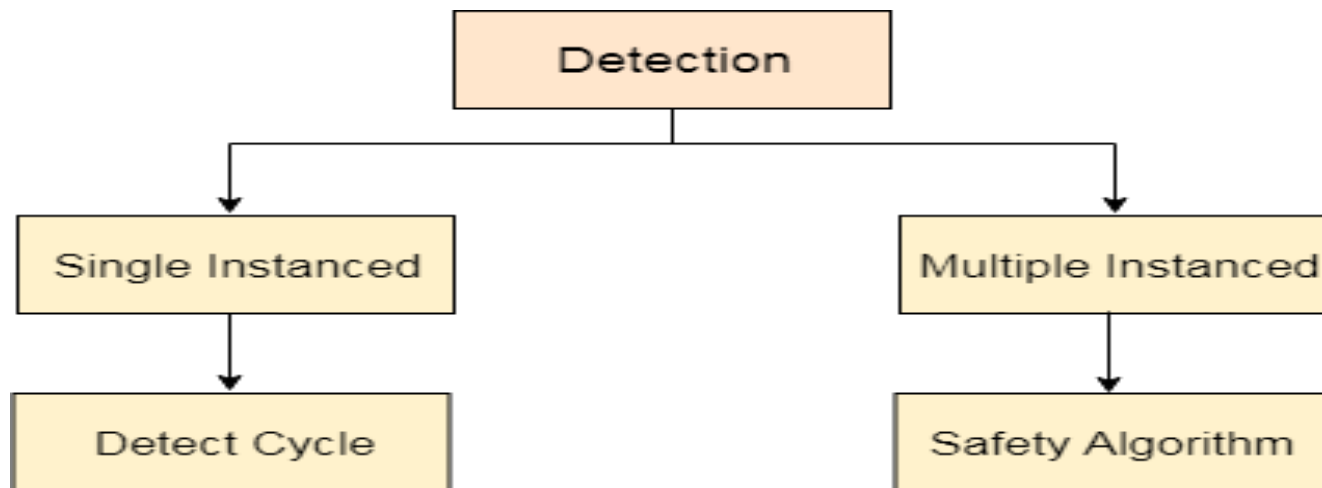
Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

$Avail = (0,0,0)$

- 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.

Operating System

Process Synchronization

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes

Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process.

These processes need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.

In order to synchronize the processes, there are various synchronization mechanisms.

Process Synchronization is mainly 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 any data at the same time.

Race Condition

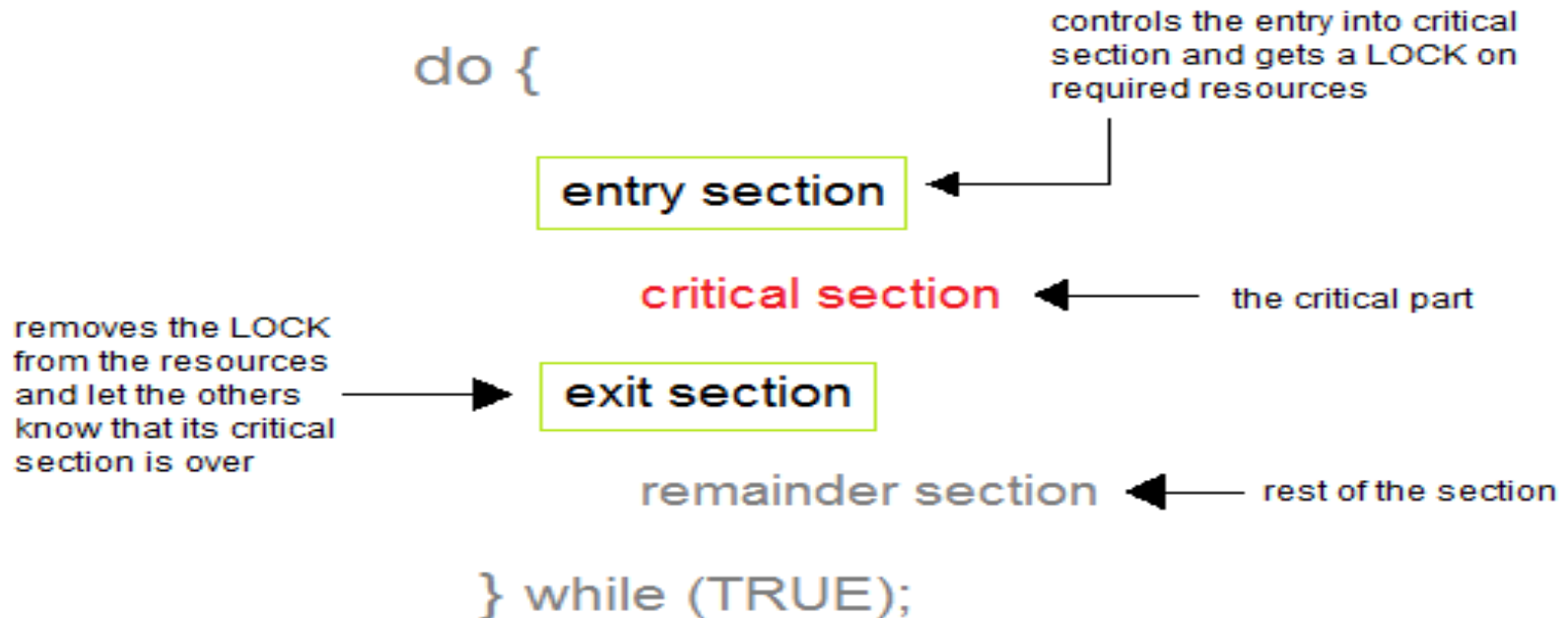
At the time when more than one process is either executing the same code or accessing the same memory or any shared variable; In that condition, there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct. This condition is commonly known as **a race condition**.

As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.

Mainly this condition is a situation that may occur inside the **critical section**. Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition in critical sections can be avoided if the critical section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

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 its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes. The entry to the critical section is mainly handled by wait() function while the exit from the critical section is controlled by the signal() function.



Entry Section

In this section mainly the process requests for its entry in the critical section.

Exit Section

This section is followed by the critical section.

The solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

Solutions for the Critical Section

The critical section plays an important role in Process Synchronization so that the problem must be solved.

Some widely used method to solve the critical section problem are as follows:

Peterson's Solution

This is widely used and **software-based solution** to critical section problems. Peterson's solution was developed by a computer scientist Peterson that's why it is named so.

With the help of this solution whenever a process is executing in any critical state, then the other process only executes the rest of the code, and vice-versa can happen.

This method also helps to make sure of the thing that only a single process can run in the critical section at a specific time.

This peterson's solution preserves all three conditions:

- **Mutual Exclusion** is comforted as at any time only one process can access the critical section.
- **Progress** is also comforted, as a process that is outside the critical section is unable to block other processes from entering into the critical section.
- **Bounded Waiting** is assured as every process gets a fair chance to enter the Critical section.

```
do
{
    Flag[i]=TRUE;

    turn=j;

while(flag[j] && turn==j);
```

Critical Section

```
Flag[i]=FALSE;
```

Remainder Section

```
}while(TRUE);
```

The above shows the structure of process **P_i** in **Peterson's solution**.

Suppose there are **N processes (P₁, P₂, ... P_N)** and as at some point of time every process requires to enter in the **Critical Section**.

A **FLAG[]** array of size N is maintained here which is by default false. Whenever a process requires to enter in the critical section, it has to set its flag as true.

Example: If P_i wants to enter it will set **FLAG[i]=TRUE**.

Another variable is called **TURN** and is used to indicate the process number that is currently waiting to enter into the critical section.

The process that enters into the critical section while exiting would change the **TURN** to another number from the list of processes that are ready.

Example: If the turn is 3 then P₃ enters the Critical section and while exiting turn=4 and therefore P₄ breaks out of the wait loop.

Synchronization Hardware

Many systems provide hardware support for critical section code.

The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time-consuming as the message is passed to all the processors.

This message transmission lag delays the entry of threads into the critical section, and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called **Mutex Locks** was introduced.

In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside the critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Classical Problems of Synchronization

Semaphore can be used in other synchronization problems besides **Mutual Exclusion**.

Below are some of the classical problem depicting flaws of process synchronization in systems where cooperating processes are present.

We will discuss the following three problems:

1. **Bounded Buffer (Producer-Consumer) Problem**
2. **Dining Philosophers Problem**
3. **The Readers Writers Problem**

1. Bounded Buffer Problem

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.

Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

In this Producers mainly produces a product and consumers consume the product, but both can use of one of the containers each time.

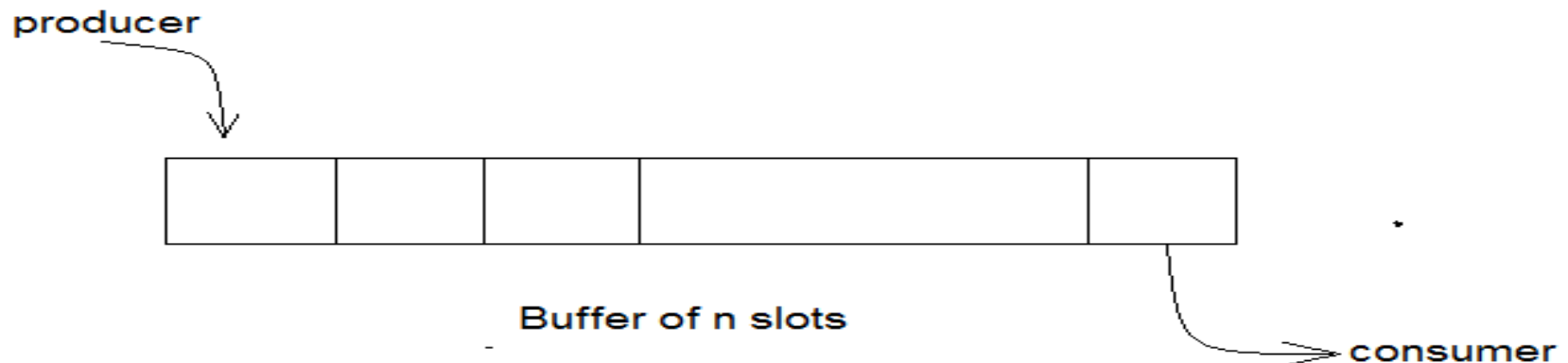
The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores.

The semaphores which will be used here are:

- **m**, a **binary semaphore** which is used to acquire and release the lock.
- **empty**, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
    // wait until empty > 0 and then decrement 'empty'
    wait(empty);
    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
    signal(mutex);
    // increment 'full'
    signal(full);
}
while(TRUE);
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
    // wait until full > 0 and then decrement 'full'
    wait(full);
    // acquire the lock
    wait(mutex);

    /* perform the remove operation in a slot */

    // release the lock
    signal(mutex);
    // increment 'empty'
    signal(empty);
}
while(TRUE);
```


- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

2. Dining Philosophers Problem

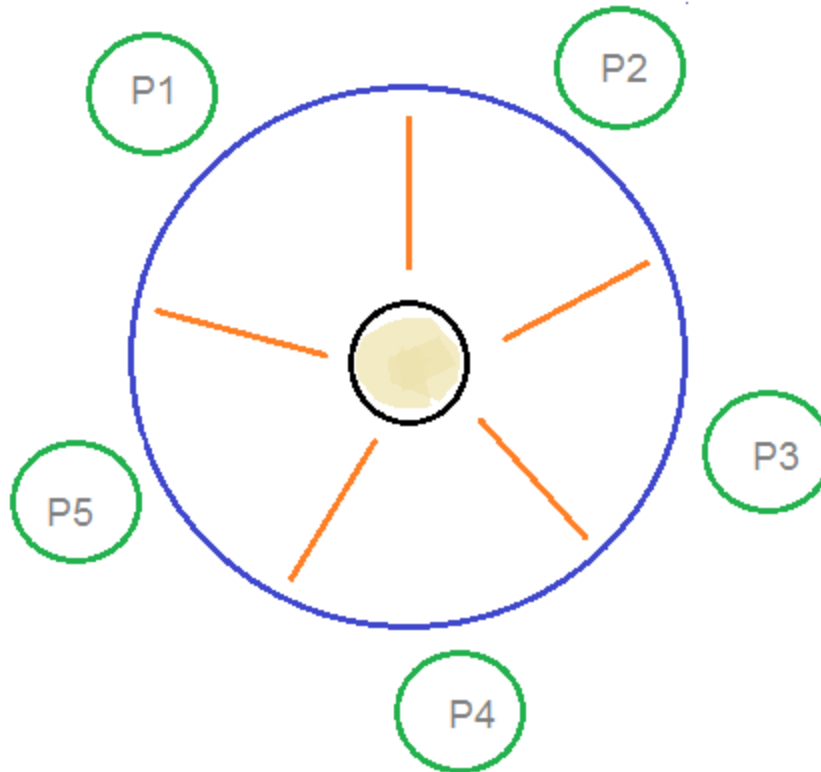
- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Dining Philosophers Problem

- The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

- At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.
- An array of five semaphores, `stick[5]`, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE) {  
    wait(stick[i]);  
    /*      mod is used because if i=5, next  
       chopstick is 1 (dining table is circular)    */  
    wait(stick[(i+1) % 5]);  
    /* eat */  
    signal(stick[i]);  
    signal(stick[(i+1) % 5]);  
    /* think */  
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

3. The Readers Writers Problem

In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.

There are various type of readers-writers problem, most centred on relative priorities of readers and writers.

The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.

What is Readers Writer Problem?

Readers writer problem is another example of a classic synchronization problem.

There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes.

There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** m and a **semaphore** w . An integer variable `read_count` is used to maintain the number of readers currently accessing the resource. The variable `read_count` is initialized to 0. A value of 1 is given initially to m and w .

Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the `read_count` variable.

The code for the **writer** process looks like this:

```
while(TRUE)
```

```
{
```

```
wait(w);
```

```
/* perform the write operation */
```

```
signal(w);
```

```
}
```

And, the code for the reader process looks like this:

```
while(TRUE) {  
    //acquire lock  
    wait(m);  
    read_count++;  
    if(read_count == 1)  
        wait(w);  
    //release lock  
    signal(m);  
    /* perform the reading operation */  
    // acquire lock  
    wait(m);    read_count--;  
    if(read_count == 0)  
        signal(w);  
    // release lock  
    signal(m);  
}
```

As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource. After performing the write operation, it increments **w** so that the next writer can access the resource.

On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.

When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.

The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.

The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.

Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.

This integer variable is called a **semaphore**.

So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by **P(S)** and **V(S)** respectively.

In very simple words, the **semaphore** is a variable that can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:

P(S): if $S \geq 1$ then $S := S - 1$
 else <block and enqueue the process>;

V(S): if <some process is blocked on the queue>
 then <unblock a process>
 else $S := S + 1$;

The classical definitions of wait and signal are:

Wait: This operation decrements the value of its argument S , as soon as it would become non-negative (greater than or equal to 1). This Operation mainly helps you to control the entry of a task into the critical section. In the case of the negative or zero value, no operation is executed. $\text{wait}()$ operation was originally termed as P ; so it is also known as **$P(S)$ operation**.

The definition of wait operation is as follows:

```
wait(S)  
{  
    while ( $S \leq 0$ ); //no operation  
    S--;  
}
```

When one process modifies the value of a semaphore then, no other process can simultaneously modify that same semaphore's value. In the above case the integer value of $S (S \leq 0)$ as well as the possible modification that is $S--$ must be executed without any interruption.

Signal: Increments the value of its argument S, as there is no more process blocked on the queue. This Operation is mainly used to control the exit of a task from the critical section. `signal()` operation was originally termed as V; so it is also known as **V(S) operation**.

The definition of signal operation is as follows:

signal(S)

{

S++;

}

Also, note that all the modifications to the integer value of semaphore in the `wait()` and `signal()` operations must be executed indivisibly.

Properties of Semaphores

- It's simple and always have a non-negative integer value.
- Works with many processes.
- Can have many different critical sections with different semaphores.
- Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types in Operating system:

Binary Semaphore:

It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**.

A binary semaphore is initialized to 1 and only takes the values 0 and 1 during the execution of a program.

In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation succeeds when the semaphore = 0. Binary Semaphores are easier to implement than counting semaphores.

Counting Semaphores:

These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Example of Use

Here is a simple step-wise implementation involving declaration and usage of semaphore.

Shared var mutex : semaphore = 1;

Process i

begin

.

.

P(mutex);

 execute CS; // cs:critical section

V(mutex);

.

.

End;

Advantages of Semaphores

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.
- No wastage of resources in semaphores because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if any condition is fulfilled in order to allow a process to access the critical section.

Disadvantages of Semaphores

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**.

Monitors in Process Synchronization

The **monitor** is one of the ways to achieve Process synchronization.

The monitor is supported by programming languages to achieve mutual exclusion between processes.

For example Java Synchronized methods.

Java provides wait() and notify() constructs.

- It is the collection of condition variables and procedures combined together in a special kind of module or a package.
- The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
- Only one process at a time can execute code inside monitors.

Syntax:

```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {...}
prodecure p2 {...}

}
```

Syntax of Monitor

Condition Variables:

Two different operations are performed on the condition variables of the monitor.

Wait.signal.

let say we have 2 condition variables

condition x, y; // Declaring variable

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation

x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

If (x block queue empty) // Ignore signal
else // Resume a process from block queue.

Advantages of Monitor:

Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor:

Monitors have to be implemented as part of the programming language . The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

Inter Process Communication (IPC)

What is Inter Process Communication?

- **Inter process communication (IPC)** is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network.
- It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.
- Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

Approaches for Inter-Process Communication



Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

Message Passing

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

It provides two operations:

- 1. Send (message)-** message size fixed or variable
- 2. Received (message)**

Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

Direct Communication:

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

Indirect Communication:

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

Shared Memory:

- Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes.
- This type of memory requires to be protected from each other by synchronizing access across all the processes.

FIFO:

- Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.