**OS**

**UNIT III**

**PROCESS COORDINATION**

**Synchronization**

Process Synchronization means coordinating the execution of processes such that no two processes access the same shared resources and data. It is required in a multi-process system where multiple processes run together, and more than one process tries to gain access to the same shared resource or data at the same time.

On the basis of 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.

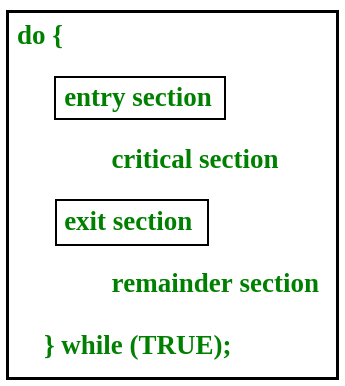
Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

**Principles of Concurrency**

Today's technology, like multi-core processors and parallel processing, allows multiple processes and threads to be executed simultaneously. Multiple processes and threads can access the same memory space, the same declared variable in code, or even read or write to the same file.

**Critical Section Problem**

Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables

.[](https://www.geeksforgeeks.org/wp-content/uploads/gq/2015/06/critical-section-problem.png)

In the entry section, the process requests for entry in the Critical Section.

Any solution to the critical section problem must satisfy three requirements:

* Mutual Exclusion : If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
* Progress : If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.
* Bounded Waiting : A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

**Semaphores**

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function.

A semaphore uses two atomic operations, wait and signal for process synchronization.  
A Semaphore is an integer variable, which can be accessed only through two operations *wait()* and *signal()*.  
There are two types of semaphores: Binary Semaphores and Counting Semaphores

* Binary Semaphores: They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.
* Counting Semaphores: They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

# Mutex vs Semaphore

What is the difference between a mutex and a semaphore? When should you use a mutex and when should you use a semaphore?

A concrete understanding of Operating System concepts is required to design/develop smart applications. Our objective is to educate the reader on these concepts and learn from other expert geeks.

As per operating system terminology, mutexes and semaphores are kernel resources that provide synchronization services (also called *synchronization primitives*). *Why do we need such synchronization primitives? Won’t only one be sufficient?*To answer these questions, we need to understand a few keywords. Please read the posts on [atomicity](https://www.geeksforgeeks.org/g-fact-57/) and [critical section](https://www.geeksforgeeks.org/g-fact-70/). We will illustrate with examples to understand these concepts well, rather than following the usual OS textual description.

**The**[**producer-consumer**](http://en.wikipedia.org/wiki/Producer-consumer_problem)**problem:**

Note that the content is a generalized explanation. Practical details vary with implementation.

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, both the threads should not run at the same time.

**Using Mutex:**

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

**Using Semaphore:**

A semaphore is a generalized mutex. In lieu of a single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

**Misconception:**

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is a binary semaphore. *But it is not*! The purpose of mutex and semaphore are different. Maybe, due to similarity in their implementation a mutex would be referred to as a binary semaphore.

Strictly speaking, a mutex is a **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal). For example, if you are listening to songs (assume it as one task) on your mobile phone and at the same time, your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

**Programming language support (MONITORS) in OS**

Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid

In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false.

**Critical Section in Synchronization**

When more than one processes access a same code segment that segment is known as critical section. Critical section contains shared variables or resources which are needed to be synchronized to maintain consistency of data variable.  
In simple terms a critical section is group of instructions/statements or region of code that need to be executed atomically ([read this post](https://www.geeksforgeeks.org/g-fact-57/) for atomicity), such as accessing a resource (file, input or output port, global data, etc.).

In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. data race across threads), the result is unpredictable.

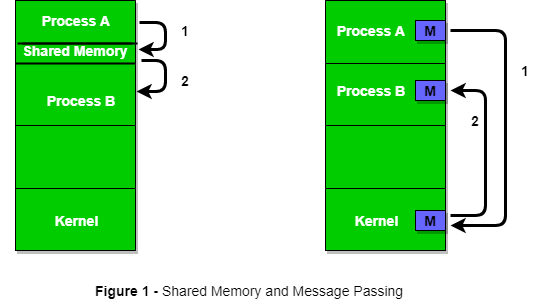
The access to such shared variable (shared memory, shared files, shared port, etc…) to be synchronized. Few programming languages have built-in support for synchronization.

It is critical to understand the importance of race condition while writing kernel mode programming (a device driver, kernel thread, etc.). since the programmer can directly access and modifying kernel data structures.

# Producer Consumer Problem using Semaphores

The Producer-Consumer problem is a classic synchronization problem in operating systems. The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process. The Producer process creates an item and adds it to the shared buffer.

An operating system can implement both methods of communication. The shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes. Let’s discuss an example of communication between processes using the shared memory method.



**The readers-writers problem**

It relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object

**Pipes**

In computer programming, especially in UNIX operating systems, a pipe is a technique for passing information from one program process to another. Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only.

Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes (inter-process communication).

**Deadlock**

Introduction to 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.

# 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 trade off 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.

## **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**

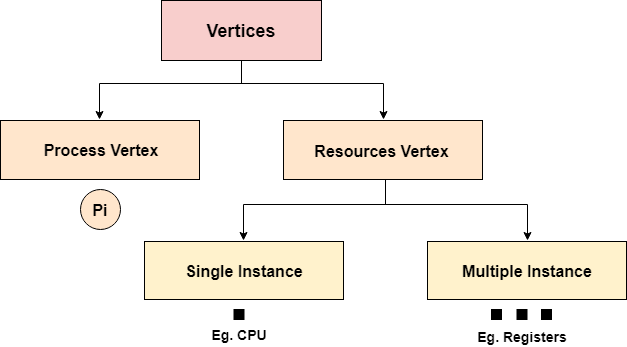
This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

# 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.

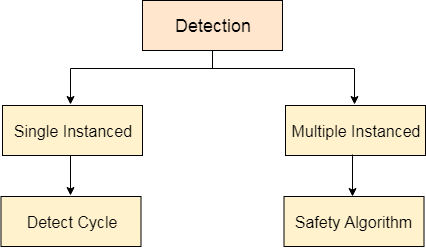


Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource

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.

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.

# Banker's Algorithm in Operating System

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the **Banker's Algorithm** in detail. Also, we will solve problems based on the **Banker's Algorithm**. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an [**operating system**](https://www.javatpoint.com/operating-system). When a new process is created in a computer system, the process must provide all types of information to the [operating system](https://www.javatpoint.com/os-tutorial) like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

## Advantages

Following are the essential characteristics of the Banker's algorithm:

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

## Disadvantages

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.
2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.
3. Each process has to know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work.Nedd[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock in a system: