**Process Synchronization**

**Definition and Purpose**

Process synchronization involves the coordination and control of concurrent processes to ensure correct and predictable outcomes. Its primary purpose is to prevent race conditions, data inconsistencies, and resource conflicts that may arise when multiple processes access shared resources simultaneously.

**Challenges in Concurrent Execution**

Concurrent execution introduces several challenges, including:

* **Race Conditions** − Concurrent processes accessing shared resources may result in unexpected and erroneous outcomes. For example, if two processes simultaneously write to the same variable, the final value may be unpredictable or incorrect.
* **Deadlocks** − Processes may become stuck in a state of waiting indefinitely due to resource dependencies. Deadlocks occur when processes are unable to proceed because each process is waiting for a resource held by another process, creating a circular dependency.
* **Starvation** − A process may be denied access to a shared resource indefinitely, leading to its inability to make progress. This situation arises when certain processes consistently receive priority over others, causing some processes to wait indefinitely for resource access.
* **Data Inconsistencies** − Inconsistent or incorrect data may occur when processes manipulate shared data concurrently. For example, if multiple processes simultaneously update a database record, the final state of the record may be inconsistent or corrupted.

**Operations on Processes**

It means the operating system puts the process from ready state into the running state. Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process.

The user can perform the following operations on a process in the operating system:

1. Process creation
2. Process scheduling or dispatching
3. Blocking
4. Preemption
5. Termination

* **Process creation**

 Process creation is the initial step to process execution. It implies the creation of a new process for execution.

**Process scheduling\dispatching**

 Scheduling or dispatching refers to the event where the OS puts the process from ready to running state. It is done by the system when there are free resources or there is a process of higher priority than the ongoing process.

* **Blocking**

Block mode is a mode where the system waits for input-output. In process blocking operation, the system puts the process in the waiting state. When a task is blocked, it is unable to execute until the task prior to it has finished using the shared resource. Examples of shared resources are the CPU, network and network interfaces, memory, and disk.

* **Preemption**

Preemption means the ability of the operating system to preempt a currently scheduled task in favour of a higher priority task. The resource being scheduled can be the processor or the I\O.

* **Termination**

Ending a process is known as process termination. There are many events that may lead to process termination, some of them are:

1. One process terminating the other process.
2. A problem in the hardware.
3. The process is fully executed, implying that the OS is finished.
4. An operating system might terminate itself due to service errors.

**Synchronization Mechanisms**

**Mutual Exclusion**

In order to avoid conflicts when processes need to use a shared resource mutual exclusion plays a crucial role in synchronizing them. Locks, semaphores and similar synchronization primitives are often utilized for ensuring exclusive access. By allowing only one process to access a shared resource at any given time, mutual exclusion prevents data races and ensures data consistency.

**Semaphores**

Semaphores are synchronization objects that maintain a count and allow or restrict access to resources based on the count value. Successful management of shared resources and coordinated process execution often require reliable tools such as semaphores. These flexible mechanisms allow for control over various resource requirements - whether binary (0 or 1) or non binary (greater than 1). Ultimately. This approach ensures efficient use of available resources without compromising other vital processes.

**Monitors**

Monitors are higher-level synchronization constructs that encapsulate shared data and the procedures that operate on them. They ensure that only one process can execute a procedure within the monitor at any given time, preventing concurrent access to shared data. Monitors provide a structured and controlled way to synchronize concurrent processes, often using condition variables to manage process coordination.

**Condition Variables**

Condition variables are synchronization primitives used in conjunction with locks or monitors to enable processes to wait for specific conditions to be satisfied before proceeding. They provide a means for processes to communicate and coordinate their actions. Processes can wait on a condition variable until another process signals or broadcasts that the condition has been met.

**Significance of Process Synchronization**

Process synchronization is crucial for the following reasons:

* **Correctness** − Synchronization mechanisms prevent race conditions and ensure the correctness of shared data. By allowing only one process to access a shared resource at a time, synchronization mechanisms maintain data integrity and consistency.
* **Resource Management** − Synchronization allows for orderly access and efficient utilization of shared resources. It ensures that processes acquire resources in a controlled manner, preventing conflicts and optimizing resource utilization.
* **Deadlock Avoidance** − Synchronization techniques help prevent and resolve deadlocks, ensuring processes can make progress. By employing deadlock prevention or handling strategies, such as resource allocation graphs or deadlock detection algorithms, deadlocks can be avoided or resolved in a timely manner.
* **Coordination** − Synchronization enables processes to coordinate their actions and communicate effectively. It provides mechanisms for processes to wait for specific conditions, signal events, and synchronize their execution, facilitating cooperation and synchronization among concurrent processes.

**Synchronization Policies and Algorithms**

Operating systems employ various synchronization policies and algorithms to ensure efficient and fair process execution. These policies determine the order in which processes are granted access to resources and influence system performance and responsiveness. For example, scheduling algorithms determine how the operating system schedules processes and assigns them CPU time, taking into account factors like process priorities and resource availability.

**Challenges and Considerations**

Deadlock Prevention and Handling

Deadlocks can pose significant challenges in process synchronization. Employing deadlock prevention or handling strategies is essential to maintain system stability. Techniques such as resource allocation graphs, deadlock detection algorithms, and deadlock avoidance strategies can be employed to prevent and resolve deadlocks effectively.

**Operations on Processes**

It means the operating system puts the process from ready state into the running state. Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process.

The user can perform the following operations on a process in the operating system:

1. Process creation
2. Process scheduling or dispatching
3. Blocking
4. Preemption
5. Termination

* **Process creation**

 Process creation is the initial step to process execution. It implies the creation of a new process for execution.

**Process scheduling\dispatching**

 Scheduling or dispatching refers to the event where the OS puts the process from ready to running state. It is done by the system when there are free resources or there is a process of higher priority than the ongoing process.

* **Blocking**

Block mode is a mode where the system waits for input-output. In process blocking operation, the system puts the process in the waiting state. When a task is blocked, it is unable to execute until the task prior to it has finished using the shared resource. Examples of shared resources are the CPU, network and network interfaces, memory, and disk.

* **Preemption**

Preemption means the ability of the operating system to preempt a currently scheduled task in favour of a higher priority task. The resource being scheduled can be the processor or the I\O.

* **Termination**

Ending a process is known as process termination. There are many events that may lead to process termination, some of them are:

1. One process terminating the other process.
2. A problem in the hardware.
3. The process is fully executed, implying that the OS is finished.
4. An operating system might terminate itself due to service errors.

**Mutual Exclusion**

A mutual exclusion (mutex) is **a program object that prevents simultaneous access to a shared resource**. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

Mutual exclusion is a property of process synchronization which states that “**no two processes can exist in the critical section at any given point of time**”.

**Race**

A race condition **occurs when two threads access a shared variable at the same time**. The first thread reads the variable, and the second thread reads the same value from the variable.

**Semaphores**

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. Semaphores are a type of synchronization primitive.

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization. The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

Semaphores are used to solve a problem of  mutual exclusion, process synchronization. There are some arguments semaphores does not gurantee freedom from race condition. Mutual exclusion is a program object that prevents simultaneous access to a shared resource.

Types of Semaphores

* Running – It states that the Process in execution.
* Ready – It states that the process wants to run.
* Idle
* Blocked
* Inactive
* Complete

Four classic synchronization problems with semaphores include:

* **Bounded–buffer (also producer-consumer or vendor-customer) problem**

The bounded-buffer problems (aka the producer-consumer problem) is a classic example of concurrent access to a shared resource. A bounded buffer lets multiple producers and multiple consumers share a single buffer. Producers write data to the buffer and consumers read data from the buffer.

* **Dining philosophers’ problem**

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers.

* **Readers and writers’ problem**

The readers-writers problem 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.

* **Sleeping barber problem.**

In computer science, the sleeping barber problem is a classic inter-process communication and synchronization problem that illustrates the complexities that arise when there are multiple operating system processes.

**Critical Region**

Sometimes a process have to access shared memory or files, or doing other critical things that can lead to races. **That part of the program which accesses the shared memory or file** is called the critical region or critical section.

**Deadlock**

A deadlock is **a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function**.

**Two processes competing for two resources in opposite order**. A single process goes through. The later process has to wait. A deadlock occurs when the first process locks the first resource at the same time as the second process locks the second resource.

Deadlock occurs **when a set of processes are in a wait state**, because each process is waiting for a resource that is held by some other waiting process. Therefore, all deadlocks involve conflicting resource needs by two or more processes.

**Deadlock Prevention**

Deadlock can be prevented by **eliminating any of the four necessary conditions, which are mutual exclusion, hold and wait, no preemption, and circular wait**. Mutual exclusion, hold and wait and no preemption cannot be violated practically. Circular wait can be feasibly eliminated by assigning a priority to each resource.

A deadlock prevention algorithm **organizes resource usage by each process to ensure that at least one process is always able to get all the resources it needs**. One such example of deadlock algorithm is Banker's algorithm.

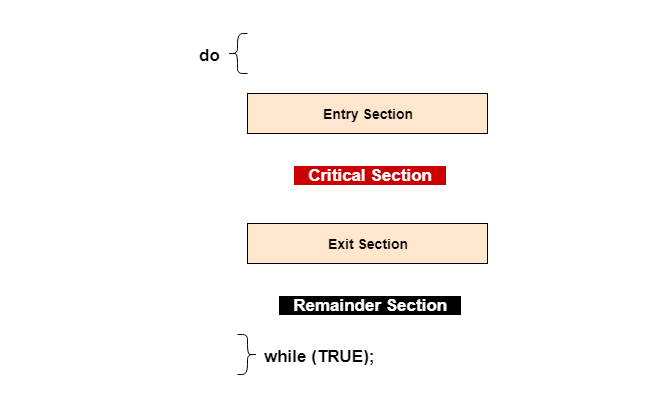
**Deadlock Avoidance**

Deadlock Avoidance is **a process used by the Operating System to avoid Deadlock**. Let's first understand what is Deadlock in Operating System. Deadlock is a situation that occurs in Operating System when any Process enters a waiting state because another waiting process is holding the demanded resource.

**Critical section**

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

A diagram that demonstrates the critical section is as follows −



In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

* **Mutual Exclusion**

Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

* **Progress**

Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

* **Bounded Waiting**

Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

**Deadlock System Model** − The **Deadlock System model** is a way to describe and analyze systems that may be prone to deadlocks, which occur when two or more processes are unable to proceed because they are each waiting for the other to release a resource. Below are the components of this model −

* **Resources** − The system has a set of **resources** that are shared among processes. These **resources** can be hardware or software components, such as memory, files, printers, or network connections. Each resource is identified by a unique name or identifier.
* **Processes** − The system has a set of **processes** that request and release resources. **Processes** are units of execution that can be started, suspended, resumed, and terminated. Each process is identified by a unique process ID.
* **Resource Allocation** − Each resource can be in one of two states , allocated or available. A resource that is allocated to a process cannot be used by any other process until it is released.
* **Request and Release** − A process can request a resource by sending a **request** to the system. If the resource is available, it will be allocated to the process. When a process is finished using a resource, it must **release** it so that it can be used by other processes.
* **Resource Dependency** − Some processes may require multiple resources to complete their tasks. A **resource dependency** graph can be used to represent the relationships between processes and resources and to detect potential deadlocks.
* **Deadlock Detection** − A deadlock can occur when two or more processes are waiting for resources that are being held by other processes, creating a circular dependency. **Deadlock detection** algorithms can be used to detect when a deadlock has occurred, so that corrective action can be taken.
* **Deadlock Resolution** − Once a deadlock has been detected, it can be resolved by breaking the circular dependency between the processes. This can be done by releasing one or more resources that are being held by a process, or by preempting one or more processes that are holding resources. The Working of some of the techniques are given below −
  + **Resource preemption** is a technique used to break the circular wait condition of a deadlock. The operating system can preempt resources from one or more processes involved in the deadlock and allocate them to the processes that need them. Preemption can be done either selectively or globally. In selective preemption, only the resources that are required to resolve the deadlock are preempted, while in global preemption, all the resources held by the deadlocked processes are preempted.
  + When a process is terminated, all the resources held by the **process are released**, and other processes can proceed. However, this approach can lead to data loss and inconsistency if the terminated process was in the middle of a critical task.
* **Deadlock Avoidance** − **Deadlock avoidance** is a technique used to prevent the occurrence of deadlocks in a computer system. The goal of deadlock avoidance is to ensure that all resources required by a process are available before the process starts execution, thereby avoiding the possibility of deadlock.

There are several algorithms that can be used for deadlock avoidance, including the banker's algorithm and the resource allocation graph. These algorithms use a mathematical model to analyze resource allocation and to determine whether a process should be allowed to start or wait for resources.

* **The banker's algorithm** is a widely used method for deadlock avoidance. It is a resource allocation algorithm that checks whether a requested resource can be granted to a process without causing a deadlock. The algorithm works by simulating the allocation of resources and checking whether a safe state can be reached. A safe state is a state where all processes can complete their execution without causing a deadlock.
* **The resource allocation graph** is another method for deadlock avoidance. It represents the allocation of resources as a directed graph. Each process is represented by a node, and each resource is represented by an edge. The algorithm checks for cycles in the graph to determine whether a deadlock has occurred. If a cycle is detected, the process requesting the resource is blocked until the required resource becomes available.

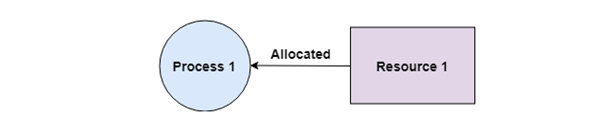
# **Deadlock Characterization**

A deadlock happens in [**operating system**](https://www.tutorialspoint.com/operating_system/index.htm) when two or more processes need some resource to complete their execution that is held by the other process.

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. They are given as follows −

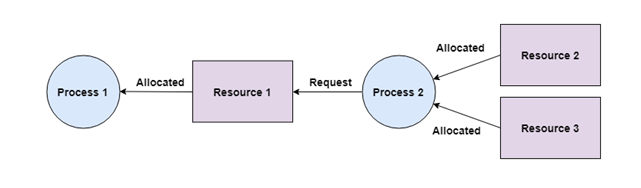
## Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



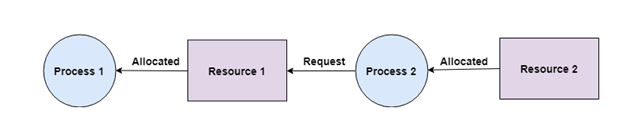
## Hold and Wait

A [**process**](https://www.tutorialspoint.com/operating_system/os_processes.htm) 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.



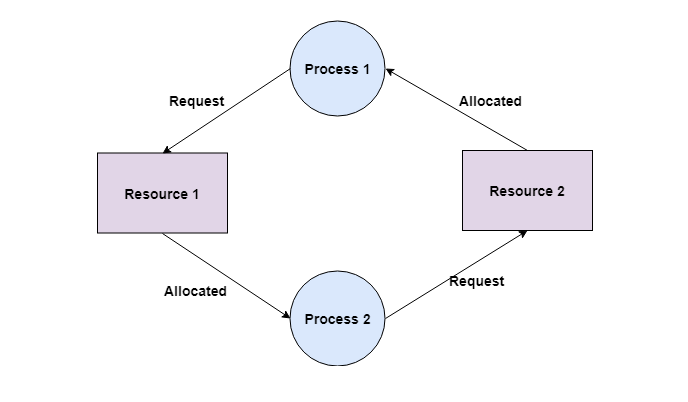
## No Preemption

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.



## 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.



# **Deadlock Avoidance**

In complex systems involving multiple processes and shared resources, the potential for deadlocks arises when processes wait for each other to release resources, causing a standstill. The resulting deadlocks can cause severe issues in computer systems, such as performance degradation and even system crashes. To prevent such problems, the technique of deadlock avoidance is employed. It entails scrutinizing the requests made by processes for resources and evaluating the available resources to determine if the grant of such requests would lead to a deadlock. In cases where granting a request would result in a deadlock, the system denies the request. Deadlock avoidance is a crucial aspect of operating system design and plays an indispensable role in upholding the dependability and steadiness of computer systems.

**Safe State and Unsafe State**

A safe state refers to a system state where the allocation of resources to each process ensures the avoidance of deadlock. The successful execution of all processes is achievable, and the likelihood of a deadlock is low. The system attains a safe state when a suitable sequence of resource allocation enables the successful completion of all processes.

Conversely, an unsafe state implies a system state where a deadlock may occur. The successful completion of all processes is not assured, and the risk of deadlock is high. The system is insecure when no sequence of resource allocation ensures the successful execution of all processes.

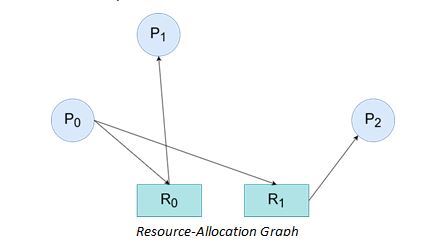
**Deadlock Avoidance Algorithms**

* When resource categories have only single instances of their resources, Resource- Allocation Graph Algorithm is used. In this algorithm, a cycle is a necessary and sufficient condition for deadlock.
* When resource categories have multiple instances of their resources, Banker’s Algorithm is used. In this algorithm, a cycle is a necessary but not a sufficient condition for deadlock.

**Resource-Allocation Graph Algorithm**

Resource Allocation Graph (RAG) is a popular technique used for deadlock avoidance. It is a directed graph that represents the processes in the system, the resources available, and the relationships between them. A process node in the RAG has two types of edges, request edges, and assignment edges. A request edge represents a request by a process for a resource, while an assignment edge represents the assignment of a resource to a process.

To determine whether the system is in a safe state or not, the RAG is analyzed to check for cycles. If there is a cycle in the graph, it means that the system is in an unsafe state, and granting a resource request can lead to a deadlock. In contrast, if there are no cycles in the graph, it means that the system is in a safe state, and resource allocation can proceed



without causing a deadlock.

The RAG technique is straightforward to implement and provides a clear visual representation of the processes and resources in the system. It is also an effective way to identify the cause of a deadlock if one occurs. However, one of the main limitations of the RAG technique is that it assumes that all resources in the system are allocated at the start of the analysis. This assumption can be unrealistic in practice, where resource allocation can change dynamically during system operation. Therefore, other techniques such as the Banker's Algorithm are used to overcome this limitation.

**Banker’s Algorithm**

The banker's algorithm is a deadlock avoidance algorithm used in operating systems. It was proposed by Edsger Dijkstra in 1965. The banker's algorithm works on the principle of ensuring that the system has enough resources to allocate to each process so that the system never enters a deadlock state. It works by keeping track of the total number of resources available in the system and the number of resources allocated to each process.

The algorithm is used to prevent deadlocks that can occur when multiple processes are competing for a finite set of resources. The resources can be of different types such as memory, CPU cycles, or I/O devices. It works by first analysing the current state of the system and determining if granting a resource request from a process will result in a safe state. A state is considered safe if there is at least one sequence of resource allocations that can satisfy all processes without causing a deadlock.

The Banker's algorithm assumes that each process declares its maximum resource requirements upfront. Based on this information, the algorithm allocates resources to each Resource-Allocation Graph process such that the total number of allocated resources never exceeds the total number of available resources. The algorithm does not grant access to resources that could potentially lead to a deadlock situation. The Banker's algorithm uses a matrix called the "allocation matrix" to keep track of the resources allocated to each process, and a "request matrix" to keep track of the resources requested by each process. It also uses a "need matrix" to represent the resources that each process still needs to complete its execution.

To determine if a request can be granted, the algorithm checks if there is enough available resources to satisfy the request, and then checks if granting the request will still result in a safe state. If the request can be granted safely, the algorithm grants the resources and updates the allocation matrix, request matrix, and need matrix accordingly. If the request cannot be granted safely, the process must wait until sufficient resources become available.

**1. Initialize the system**

* Define the number of processes and resource types.
* Define the total number of available resources for each resource type.
* Create a matrix called the "allocation matrix" to represent the current resource allocation for each process.
* Create a matrix called the "need matrix" to represent the remaining resource needs for each process.

**2. Define a request**

* A process requests a certain number of resources of a particular type.

**3. Check if the request can be granted**

* Check if the requested resources are available.
* If the requested resources are not available, the process must wait.
* If the requested resources are available, go to the next step.

**4. Check if the system is in a safe state**

* Simulate the allocation of the requested resources to the process.
* Check if this allocation results in a safe state, meaning there is a sequence of allocations that can satisfy all processes without leading to a deadlock.
* If the state is safe, grant the request by updating the allocation matrix and the need matrix.
* If the state is not safe, do not grant the request and let the process wait.

**Release the Resources**

* When a process has finished its execution, releases its allocated resources by updating the allocation matrix and the need matrix.

The above steps are repeated for each resource request made by any process in the system. Overall, the Banker's algorithm is an effective way to avoid deadlocks in resource constrained systems by carefully managing resource allocations and predicting potential conflicts before they arise.

# **Deadlock Prevention**

Deadlock prevention is a technique used in computer science to avoid situations where multiple processes or threads are blocked and unable to proceed because they are waiting for each other to release resources that they need to complete their tasks. Deadlocks can be detrimental to the performance of a system and can cause significant delays in completing tasks. To prevent deadlocks, various techniques such as resource allocation graph, preemptive scheduling, and detection and recovery are used. Each of these techniques helps in ensuring that resources are used efficiently and prevent situations where a deadlock may occur, thereby ensuring optimal performance of the system.

### Definition of Deadlock Prevention

Deadlock prevention is a technique used in computer science to avoid situations where multiple processes or threads are blocked and unable to proceed because they are waiting for each other to release resources that they need to complete their tasks. The main goal of deadlock prevention is to ensure that resources are used efficiently and that situations where a deadlock may occur are avoided. By implementing appropriate techniques for preventing deadlocks, computer systems can improve their efficiency, reduce delays, and avoid the risk of system failure caused by deadlocks. Deadlock prevention techniques may include resource allocation graph, pre-emptive scheduling, detection and recovery, or a combination of these methods.

### Importance of Preventing Deadlocks

Preventing deadlocks is crucial for ensuring the smooth operation and optimal performance of computer systems. Deadlocks occur when multiple processes or threads are blocked and unable to proceed because they are waiting for each other to release resources that they need to complete their tasks. This can result in significant delays in completing tasks, reduced system performance, and even system failure. Deadlocks can be especially problematic in real-time systems, where delays can have severe consequences. Therefore, preventing deadlocks is essential to ensure that computer systems operate efficiently, effectively, and reliably. By implementing appropriate techniques for preventing deadlocks, computer systems can improve their performance, reduce delays, and avoid the risk of system failure caused by deadlocks.

### Resource Allocation Graph

Resource allocation graph is a technique used for preventing deadlocks in computer systems. In this technique, resources are represented as nodes in a directed graph, and the edges represent the requests for resources. There are two types of edges in the graph - request edges and assignment edges. A request edge points from a process to a resource that the process needs, while an assignment edge points from a resource to a process that has been allocated that resource.

### Detection of cycles in the graph Explanation of resource allocation graph

Deadlocks can occur in the resource allocation graph when there is a cycle in the graph. A cycle in the graph indicates that a process is waiting for a resource that is held by another process that is also waiting for a resource held by the first process. This situation creates a deadlock, and no process can proceed further.

### Prevention of deadlocks using resource allocation graph

To prevent deadlocks using the resource allocation graph technique, the system must ensure that no cycles exist in the graph. If a cycle exists, it means that a deadlock may occur, and the system must take appropriate steps to break the cycle. This can be done by either releasing a resource or pre-emptively removing a resource from a process that is holding it. By breaking the cycle, the system ensures that no deadlock can occur.

In summary, the resource allocation graph is a useful technique for preventing deadlocks in computer systems. It allows the system to monitor the use of resources and detect potential deadlocks by analysing the graph for cycles. By taking appropriate steps to break the cycle, the system can avoid the occurrence of deadlocks and ensure the smooth operation of the system.

### Pre-emptive Scheduling

Pre-emptive scheduling is a technique used for preventing deadlocks in computer systems. In this technique, the system is designed to pre-empt a process that is holding a resource for too long. When a process is pre-empted, the system forces the process to release the resource, which can then be allocated to another process that needs it.

## Explanation of pre-emptive scheduling

To identify processes that are holding resources for too long, the system can use various techniques such as monitoring the waiting time of processes, tracking the use of resources, and analysing the behaviour of processes. Once the system identifies a process that is holding a resource for too long, it can pre-empt that process and force it to release the resource.

### Forced release of resources to prevent deadlocks

Forced release of resources can prevent deadlocks by ensuring that resources are not held for extended periods. When a resource is released, it can be allocated to another process that needs it, thereby preventing a situation where multiple processes are blocked and unable to proceed.

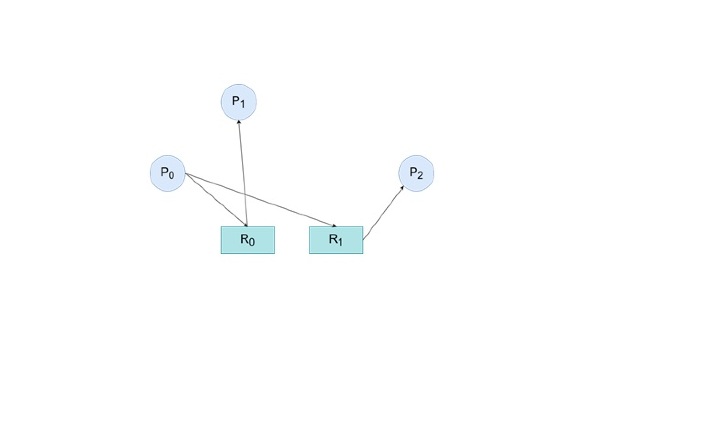
Pre-emptive scheduling can be an effective technique for preventing deadlocks, but it may also introduce some overhead and may not be suitable for all systems. Additionally, pre-emptive scheduling may impact the performance of the system if too many processes are pre-empted, and resources are frequently released and allocated. Therefore, it is important to use pre-emptive scheduling judiciously and evaluate its impact on the system's performance.

## Deadlock Detection

Deadlock detection is a crucial task in computer systems that utilize shared resources. It involves identifying and flagging situations where two or more processes are blocked, unable to proceed because they are waiting for resources that are being held by other processes. When a deadlock is detected, the system can initiate recovery procedures to break the deadlock and restore the system to a functional state. It is essential to monitor and detect deadlocks as early as possible to prevent any negative impact on the system's performance and stability. A delay in detecting deadlocks can result in significant wait times for processes and unresponsive systems, leading to user frustration and potential business losses. There are two primary methods for detecting deadlocks: resource allocation graph (RAG) and wait-for graph (WFG).

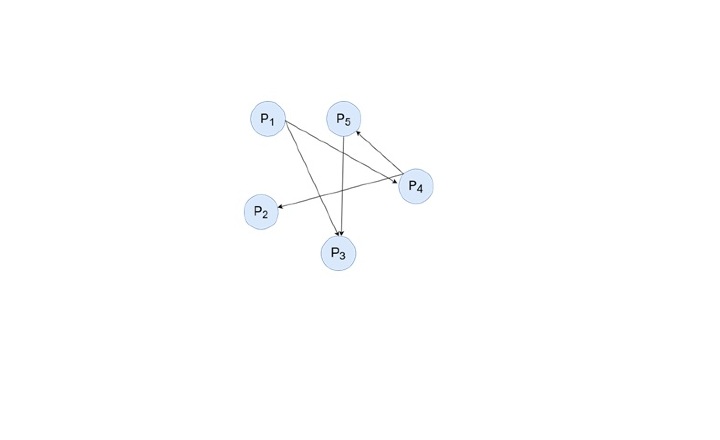
## Resource Allocation Graph (RAG)

The resource allocation graph (RAG) is a widely used method for deadlock detection in computer systems. The RAG is a graphical representation of the current allocation state of resources and the processes that are holding them. The nodes of the graph represent the resources and the processes, and the edges represent the allocation relationship between them. In the RAG method, a cycle in the graph indicates the presence of a deadlock. When a cycle is detected, it means that each process in the cycle is holding on to at least one resource that is required by another process in the cycle, resulting in a deadlock. The RAG method is highly efficient and can quickly detect deadlocks, making it an essential technique in modern operating systems.



## Wait-For Graph (WFG)

The wait-for graph (WFG) is a common method used in deadlock detection in computer systems. The WFG is a graphical representation of the dependencies between the processes and the resources that they are waiting for. In the WFG, the nodes represent processes, and the resources are represented as edges. Each edge points from the process that is waiting for a resource to the process that currently holds that resource. The WFG method can efficiently detect deadlocks by analysing the graph for cycles. If a cycle is found, it indicates that a set of processes is waiting for resources that are being held by other processes in the same set, resulting in a deadlock. The system can then take appropriate actions to break the deadlock, such as rolling back the transactions or aborting some of the processes.



## Deadlock Recovery

Deadlock recovery is a critical process that is initiated after a deadlock has been detected in a computer system. This complex process involves a set of actions and procedures that are undertaken to resolve the deadlock by breaking the cycle of resource dependency between the processes involved in the deadlock. The recovery process usually entails the identification of the process or processes that are causing the deadlock and releasing one or more of the resources that they are holding. This involves a careful analysis of the system's state to determine which resources can be safely released without compromising the system's integrity. There are four primary methods of deadlock recovery: process termination, resource pre-emption, priority inversion, and rollback.

## Process Termination

Process termination is a simple method for resolving deadlocks. In this method, the operating system identifies the processes involved in the deadlock and terminates one or more processes. This frees up the resources held by the terminated processes, which can be used by the remaining processes to continue their execution. However, this method has some drawbacks, such as loss of data, abrupt termination of processes, and inconsistency in the system.

## Resource Pre-emption

Resource pre-emption is a more complex method for resolving deadlocks. In this method, the operating system identifies the resources involved in the deadlock and selects one or more resources to be pre-empted. The resources are then taken away from the process holding them and allocated to the waiting processes. The pre-empted process is suspended until the required resources become available again. This method can cause delays in the execution of the pre-empted process and can result in a suboptimal allocation of resources.

## Priority Inversion

Priority inversion is a method for resolving deadlocks in real-time systems. In this method, the priority of the processes is changed to avoid deadlock situations. The process holding the required resources is given a higher priority, and the process waiting for the resources is given a lower priority. This method can lead to the inversion of priorities, which can cause performance issues and degrade the performance of the system. Additionally, this method can also lead to starvation of lower priority processes, as higher priority processes can keep pre-empting the resources.

## Rollback

Rollback is a method for resolving deadlocks that is commonly used in database systems. In this method, the system rolls back the transactions of the involved processes to a previous state where they were not deadlocked. This method requires that the system maintains a log of all the transactions and the state of the system at different points in time. The system can then roll back the transactions to the previous state and re-execute them. This method can cause significant delays in the execution of the transactions and can result in a loss of data.

.