**Unit 4 : Inter-process Communication and Synchronization**

* **Need for Inter-process Communication (IPC):**

**Data Sharing**: Processes often need to share data or information with each other. This could be for collaboration or for passing data from one program to another. IPC mechanisms provide a way for processes to exchange data.

**Resource Sharing:** Multiple processes may need to access shared resources, such as a printer, database, or network connection. IPC helps in coordinating access to these shared resources to prevent conflicts.

**Modularity:** Large applications are often divided into smaller, modular processes or threads. These modules need to communicate with each other to perform a coordinated task.

***Example of IPC:***

1. **Pipes**: In Unix-like systems, pipes allow data to be passed from one process to another. For example, you can use a pipe to pass the output of one program as input to another program.
2. **Message Queues**: Processes can send messages to each other through message queues. For example, a server process can receive requests from multiple client processes through message queues.
3. **Shared Memory**: Processes can map a regi+on of memory that is shared between them. This allows them to read and write data in a shared memory location. For instance, a producer-consumer problem can be solved using shared memory, where one process produces data, and another consumes it.

* **Need for Synchronization:**

1. **Preventing Race Conditions**: When multiple processes access shared resources concurrently, race conditions can occur, leading to unpredictable and undesirable behavior. Synchronization mechanisms ensure that only one process can access the resource at a given time.
2. **Ordering and Coordination**: In some cases, processes need to execute in a specific order or coordinate their activities. Synchronization primitives like semaphores and barriers enable processes to wait for a certain condition before proceeding.
3. **Avoiding Deadlocks**: Synchronization also helps in avoiding deadlock situations where processes are waiting indefinitely for a resource that will never be released.

* ***Examples of Synchronization:***

1. **Murexes (Mutual Exclusion)**: Mutex locks allow only one process to access a shared resource at a time. For example, in a multi-threaded program, a mutex can protect access to a critical section of code.
2. **Semaphores**: Semaphores are used for counting and synchronization. They can be used to control access to a fixed number of resources, like limiting the number of concurrent connections to a server.
3. **Barriers**: A barrier synchronization mechanism is used when a group of processes need to wait for all of them to reach a certain point before proceeding. For instance, in parallel computing, processes might need to synchronize at the end of a parallel task before continuing with the next step.

In summary, IPC and synchronization mechanisms are crucial for ensuring that multiple processes can work together effectively, share resources safely, and coordinate their activities in a way that avoids conflicts and race conditions. These concepts are fundamental in the design of concurrent and multi-process systems.

* **Mutual Exclusion (Mutex)**

Mutual exclusion is a fundamental concept in computer science and concurrent programming. It refers to the property or mechanism that ensures that only one process or thread can access a shared resource, such as a variable, file, or data structure, at any given time. This exclusivity prevents multiple processes or threads from simultaneously modifying the shared resource, which can lead to data corruption or unpredictable behavior.

In essence, mutual exclusion guarantees that when one process is actively using the shared resource, all other processes must wait until it releases the resource before they can access it. This synchronization mechanism is essential to prevent race conditions and maintain data integrity in multi-threaded or multi-process environments.

One common way to implement mutual exclusion is through the use of mutexes (short for "mutual exclusion locks"). A mutex acts as a gatekeeper for the shared resource, allowing only one thread or process to enter a critical section of code at a time. When a thread or process acquires the mutex, it gains exclusive access to the resource, and all other threads or processes attempting to acquire the same mutex are blocked until the resource is released by the owning thread or process.

Mutexes are just one example of how mutual exclusion can be achieved, but the core idea remains the same: ensuring that shared resources are accessed in a controlled and exclusive manner to prevent conflicts and maintain data consistency

* **Here's how mutual exclusion works:**

**Critical Section:** The critical section is a part of the code that accesses shared resources or data that should be protected from concurrent access. Only one process or thread should be allowed to execute the critical section at any given time to maintain data integrity.

**Mutex Lock:** A mutex (short for "mutual exclusion") is a synchronization primitive that is associated with the critical section. When a process or thread wants to enter the critical section, it first tries to acquire the mutex lock.

**Acquiring the Lock:** If the mutex lock is available (i.e., not held by another process or thread), the requesting process or thread acquires the lock and is allowed to enter the critical section.

**Releasing the Lock:** When a process or thread is done with the critical section, it releases the mutex lock, allowing other processes or threads to acquire it and enter the critical section.

* **What is a Semaphore?**

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.

The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

A semaphore is a special kind of synchronization data that can be used only through specific synchronization primitives. When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0. If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

* ***Operations in Semaphores***

Wait() and signal() are the two basic operations used to manipulate semaphores in an operating system.

1. **Wait():** When a process or thread performs a wait() operation on a semaphore, it checks the current value of the semaphore. If the value is positive, the process or thread acquires the semaphore and decrements its value. If the value is zero, the process or thread is blocked and added to a queue of waiting processes until the semaphore’s value becomes positive.

***Syntax of Wait():***

wait(S)

{

while (S<=0);

S--;

}

1. **Signal():** When a process or thread performs a signal() operation on a semaphore, it increments the semaphore's value. If there are any processes or threads waiting on the semaphore, one of them is unblocked and allowed to acquire the semaphore.

***Syntax for Wait():***

signal(S)

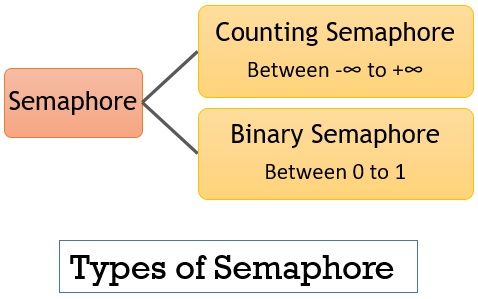
{

S++;

}

* **Types of Semaphores in OS**

There are two types of semaphores:

****

* **Counting Semaphore:**

The counting semaphores controls access to resources that have finite instances. Therefore the value of counting semaphore is not restricted to a certain domain.

The counting semaphore is initialized with a value equivalent to the number of resources available. When a process wants to access the resource it executes the entry section code where the wait() operation decrements the value of semaphore and allot the resource to the process.

When the counting semaphore value decreases to 0 it means no resources are available now and the process that executes the entry section code further would be blocked. Now when a process releases a resource it executes its exit section code where the signal() operation would increment the value of counting semaphore and release a blocked process which can further try to execute the entry section code.

* **Binary Semaphores:**

Binary semaphores are generally used to access the critical section. As we know that only one process can enter a critical section at a time therefore the value of binary semaphores ranges over 0 to 1.

Consider that the binary semaphore value is initialized to 1. Now if a process P1 want to enter the critical section then it executes the entry section code where the wait() operation decrements the semaphore value to 0 and enters the critical section. Now when second process P2 tries to enter the critical section then it executes the entry section code where the wait() operation verifies the binary semaphore value is already 0 which means no process further can enter the critical section and blocks the second process P2.

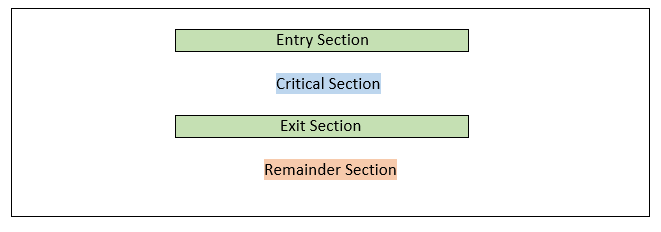
When process P1 leaves the critical section it executes the signal() operation of exit section which increments the semaphore value to 1 and releases a blocked process P2  which can further try to execute entry section code to enter the critical section.

* **What is a Critical Section Problem?**

The Critical Section Problem is a Code Snippet. This code snippet contains a few variables. These variables can be accessed by a few processes. There is a condition for these processes.

The condition is that only one process can only enter the critical section. Remaining Processes which are interested to enter the critical section have to wait for the process to complete its work and then enter the critical section.

* **Critical Section Representation**



* ***Advantages of Semaphores in OS***
* Enforce mutual exclusion to prevent race conditions.
* Synchronize process execution.
* Prevent deadlocks.
* Efficiently manage system resources.
* ***Disadvantages of Semaphores in OS***
* Potential for deadlock if not used properly.
* Overuse of semaphores can lead to complex and hard-to-maintain code.
* Can be difficult to debug when synchronization issues arise.
* Not suitable for all types of synchronization problems.
* **Characteristics of semaphore**

Here are the key characteristics of semaphores in operating systems:

1. **Integer Value:** Semaphores are represented as integer variables. The value of a semaphore can be non-negative, typically initialized to a non-negative integer.
2. **Atomic Operations:** Semaphore operations, such as incrementing (V or signal) and decrementing (P or wait), are atomic. This means that they are uninterruptible and cannot be split into multiple steps, ensuring consistency in multi-process or multi-threaded environments.
3. **Counting Semaphores:** Counting semaphores can have a value greater than or equal to zero. They are used to control access to multiple instances of a shared resource. For example, if you have five printers in a computer lab, a counting semaphore with an initial value of 5 can ensure that no more than five processes can access the printers simultaneously.
4. **Binary Semaphores:** Binary semaphores can only have two possible values: 0 and 1. They are commonly used for mutual exclusion, where the semaphore is initially set to 1. A binary semaphore is often called a mutex (short for mutual exclusion).
5. **Blocking and Unblocking:** When a process or thread attempts to decrement (P or wait) a semaphore whose value is 0, it blocks (waits) until the semaphore becomes greater than 0. When a semaphore is incremented (V or signal), a blocked process or thread may be unblocked if it was waiting on that semaphore.
6. **Resource Synchronization**: Semaphores are used to control access to shared resources to avoid race conditions and data corruption. They help ensure that only one process or thread accesses a critical section (shared resource) at a time.
7. **Deadlock Prevention**: Semaphores can be used to prevent deadlock situations by properly managing the order of acquiring and releasing semaphores. Care must be taken to avoid circular dependencies.
8. **Initialization**: Semaphores are typically initialized to a specific value before being used. The initial value determines the number of processes or threads that can access the resource concurrently.
9. **Priority Inversion:** In some cases, semaphores may lead to priority inversion, where a lower-priority process holds a semaphore needed by a higher-priority process. Operating systems may use priority inheritance or priority ceiling protocols to mitigate this issue.
10. **Inter-Process/Thread Communication**: Besides resource synchronization, semaphores can be used for inter-process or inter-thread communication by allowing processes or threads to signal each other when certain conditions are met.

* **Busy-wait Implementation**

A busy-wait implementation, also known as spinning or polling, is a synchronization technique where a process or thread repeatedly checks a specific condition or flag in a loop without yielding the CPU or entering a blocking state.

Instead of waiting for the condition to become true, the process or thread actively consumes CPU cycles by continuously reevaluating the condition. This approach is suitable for situations where the expected waiting time for the condition to be met is very short.

It offers low and predictable latency because there is no context switching or thread scheduling overhead. However, it can be inefficient and CPU-intensive if the waiting time is long, as it keeps the CPU busy unnecessarily. Busy-wait implementations are often used in scenarios like spinlocks or when waiting for hardware devices to complete operations.

***Example:***

* A common example of busy-wait is spinlocks. In a multi-threaded program, when one thread wants to access a critical section of code, it checks whether the spinlock associated with that critical section is locked. If it is locked (i.e., the condition is not met), the thread continuously checks until the spinlock is released (i.e., the condition becomes true).
* Another example is waiting for a hardware device to complete an operation. A device driver might use busy-wait to repeatedly poll the device's status register until it indicates that the operation is complete.
* ***Advantages:***
* Low Overhead: Busy-wait has low overhead compared to blocking and unblocking processes or threads, as it doesn't involve context switching or thread scheduling.
* Predictable Latency: Busy-wait can provide low and predictable latency because there is no waiting for the operating system scheduler to allocate CPU time.
* ***Disadvantages:***
* Wasteful of CPU: Busy-wait consumes CPU cycles actively checking the condition, which can be inefficient if the waiting time is long.
* Can Lead to Priority Inversion: In multi-threaded environments, busy-wait can lead to priority inversion, where lower-priority threads preempt higher-priority threads because they are continuously running and holding resources.
* **Queuing implementation of semaphore**

In operating systems, a semaphore is a synchronization primitive used to control access to shared resources in a multi-process or multi-threaded environment. Semaphores can be implemented in various ways, and one common implementation is a queuing mechanism, where processes or threads requesting the semaphore are placed in a queue until they can acquire the semaphore. Here's an explanation of the queuing implementation of a semaphore in an operating system context:

1. **Initialization:**

To implement a semaphore using a queue, you typically start by initializing the semaphore with an initial count and an empty queue. The initial count represents the number of available resources.

1. **Acquiring the Semaphore:**
   * When a process or thread wants to access the shared resource protected by the semaphore, it first tries to acquire the semaphore.
   * If the semaphore's count is greater than zero (i.e., there are available resources), the process or thread decrements the count, indicating that it has successfully acquired a resource, and proceeds with its work.
   * If the count is zero (i.e., no available resources), the requesting process or thread is blocked and added to the queue of waiting processes or threads.
2. **Releasing the Semaphore:**
   * When a process or thread finishes using the shared resource, it releases the semaphore.
   * If there are processes or threads waiting in the queue, one of them is dequeued and allowed to proceed. The count is incremented to reflect the release of a resource.
   * If the queue is empty, the count is simply incremented.

* **Producer-Consumer problem**

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one processes.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

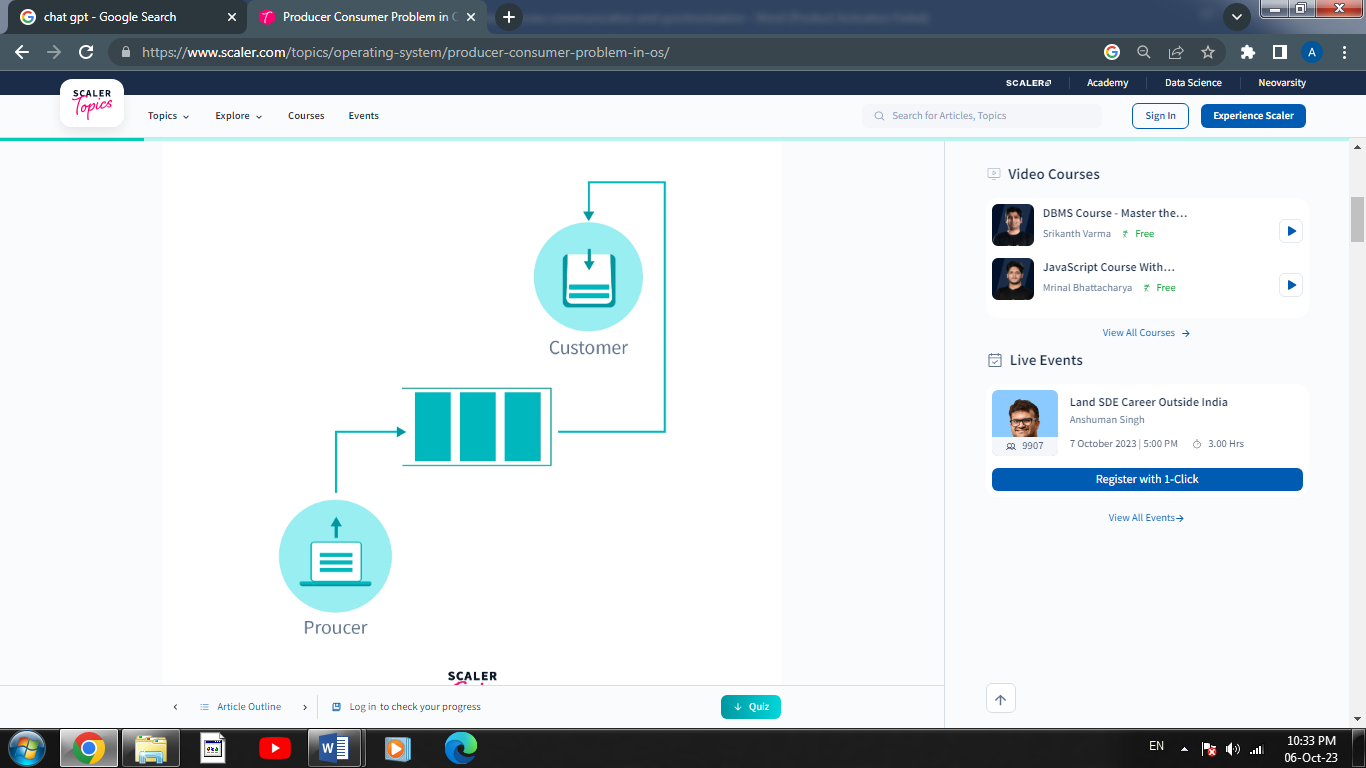
* **What is Producer Consumer Problem?**

Before knowing what is Producer-Consumer Problem we have to know what are Producer and Consumer.

In operating System Producer is a process which is able to produce data/item.

Consumer is a Process that is able to consume the data/item produced by the Producer.

Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.



So, what are the Producer-Consumer Problems?

Producer Process should not produce any data when the shared buffer is full.

Consumer Process should not consume any data when the shared buffer is empty.

The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.

* **Solution For Producer Consumer Problem**

To solve the Producer-Consumer problem three semaphores variable are used :

Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

**Full :** The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.

**Empty** The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the **BUFFER-SIZE** as initially, the whole buffer is empty.

**Mutex :** Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.

* **Critical Region and Conditional Critical Area**

# **Critical Regions:**

In an operating system, a critical region refers to a section of code or a data structure that must be accessed exclusively by one method or thread at a time. Critical regions are utilized to prevent concurrent entry to shared sources, along with variables, information structures, or devices, that allow you to maintain information integrity and keep away from race conditions.

The concept of important regions is carefully tied to the want for synchronization and mutual exclusion in multi-threaded or multi-manner environments. Without proper synchronization mechanisms, concurrent admission to shared resources can lead to information inconsistencies, unpredictable conduct, and mistakes.

To implement mutual exclusion and shield important areas, operating structures provide synchronization mechanisms, inclusive of locks, semaphores, or monitors. These mechanisms ensure that the handiest one procedure or thread can get the right of entry to the vital location at any given time, even as other procedures or threads are averted from entering till the cutting-edge occupant releases the lock.

The critical section problem needs a solution to synchronise 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.

* **Progresss**

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 Waitings**

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

# **Conditional Critical Area:**

The term "Conditional Critical Area" is not a standard concept or terminology in the field of operating systems. It's possible that it's a term used in a specific context or by a particular organization, but it doesn't have a widely recognized definition in the field of operating systems as of my last knowledge update in September 2021.

In the context of operating systems, critical areas typically refer to sections of code or data that require special handling, such as protection from concurrent access by multiple processes or threads. These critical areas are often protected using synchronization mechanisms like locks, semaphores, or mutexes to ensure that only one process or thread can access them at a time, preventing race conditions and ensuring data consistency.

If you have more specific information or context about what you mean by "Conditional Critical Area" or if it's related to a specific operating system or technology introduced after my last knowledge update, I would be happy to provide more information based on that context.

* ***Some key point of Critical Region and Condition Critical Area***

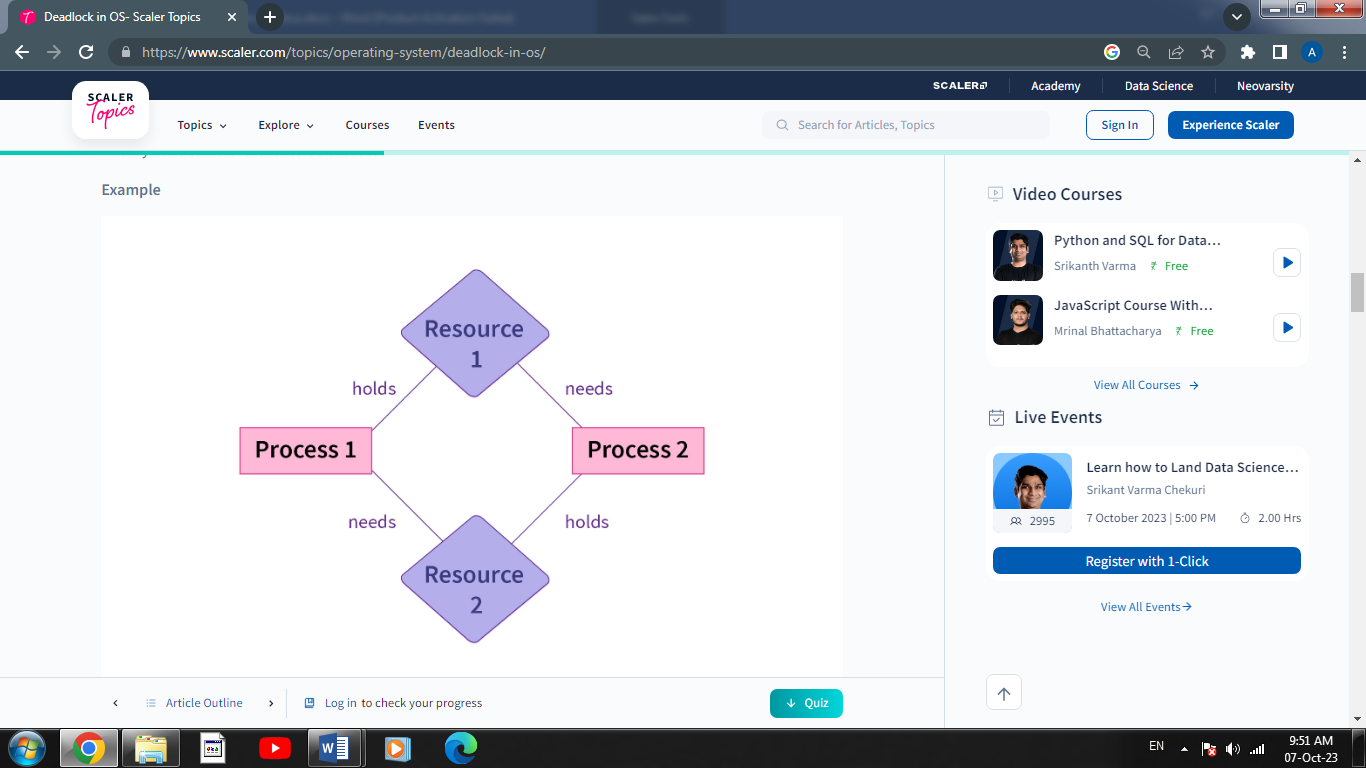
1. **Critical Region:**
   * A Critical Region, often referred to as a Critical Section, is a specific section of code or a block of code within a program that accesses shared resources.
   * In a multi-threaded or multi-process environment, only one thread or process should be allowed to execute the code within the Critical Region at a given time.
   * The primary purpose of defining a Critical Region is to prevent data corruption or inconsistencies caused by concurrent access to shared resources. This is typically achieved using synchronization mechanisms like locks, semaphores, or mutexes.
   * When a thread or process enters a Critical Region, it acquires the associated synchronization mechanism, which prevents other threads or processes from entering the same Critical Region until the synchronization mechanism is released.
2. **Conditional Critical Area:**
   * A Conditional Critical Area is an extension of the concept of a Critical Region. It refers to a section of code or a block of code that may or may not need to be executed based on certain conditions.
   * Unlike a regular Critical Region, a Conditional Critical Area might be entered by multiple threads or processes concurrently, but it should only execute its critical code when specific conditions are met.
   * Conditional Critical Areas are typically protected using synchronization mechanisms just like regular Critical Regions. However, the synchronization mechanism may be used to check the conditions before allowing access to the critical code within the area.
   * The goal of a Conditional Critical Area is to ensure that when multiple threads or processes attempt to access it concurrently, only the one that satisfies the conditions can execute the critical code.

* **What is Deadlock in OS?**

## 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.

All the processes in a system require some resources such as a central processing unit(CPU), file storage, input/output devices, etc. to execute it. Once the execution is finished, the process releases the resource it was holding. However, when many processes run on a system they also compete for the resources they require for execution. This may cause a deadlock situation.

A **deadlock** is a situation in which more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process. Therefore, none of the processes gets executed.



In the above figure, there are two processes and two resources. Process 1 holds "Resource 1" and needs "Resource 2" while Process 2 holds "Resource 2" and requires "Resource 1". This creates a situation of deadlock because none of the two processes can be executed

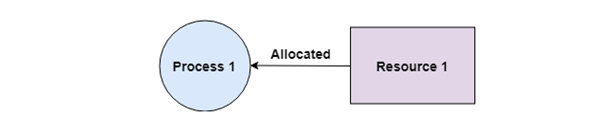
* **Deadlock Prevention**

Deadlock prevention is a set of strategies and techniques employed in operating systems and concurrent programming to proactively avoid the occurrence of deadlocks, rather than dealing with them after they have already happened. A deadlock is a situation in which two or more processes or threads are unable to proceed because they are each waiting for a resource that is held by another, resulting in a standstill in the system.

The primary goal of deadlock prevention is to design the system or application in such a way that it becomes impossible for the conditions necessary for a deadlock to occur to ever be met. This typically involves setting up rules and constraints for how processes or threads can request and access resources, ensuring that these rules are followed to eliminate the possibility of circular waits, resource conflicts, and other conditions that lead to deadlocks.

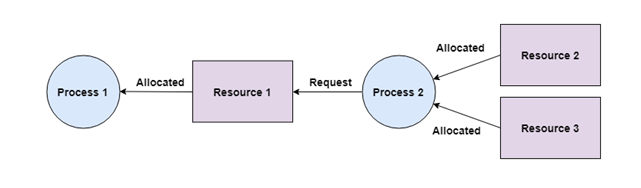
* **Necessary Conditions for Deadlock**

1. **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.

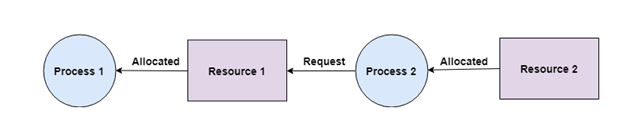
1. **Hold and Wait**

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.



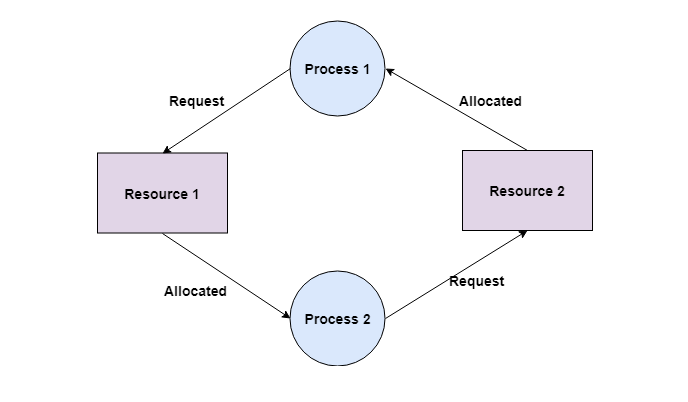
1. **No Preemption**

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.



1. **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 OS**

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.

1. **Wait/Die:** In this method, a process that requests a resource that is not available will wait until the resource becomes available. If a process that holds the resource is also requesting a resource that is not available, the process that has been waiting the longest will be allowed to proceed.
2. **Wound/Wait:** In this method, a process that requests a resource that is not available will be killed (wounded) if a process that holds the resource is also requesting a resource that is not available. The process that was killed will have to request the resource again later.
3. **Resource allocation graph:** By representing the resources and the processes as nodes in a graph, the operating system can use algorithms to check for safe states before allocating resources.
4. **Banker’s Algorithm:** This algorithm is a variant of Resource Allocation Graph algorithm, it uses the available and maximum resource information for each process, the algorithm checks for safe state before allocating resources.
5. **Time-stamp ordering:** In this method, a process is only allowed to request a resource if its time stamp is greater than the time stamp of the process that holds the resource.

These are some of the common methods used in operating systems for deadlock avoidance. The best method to use depends on the specific requirements of the system and the resources being used.

* **Banker’s Algorithm**

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

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.

***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.  
   **Need[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.

* **Safe sate and unsafe state**

1. **Safe State:**
   * A system is in a safe state if there exists a sequence of processes that can be executed without causing a deadlock.
   * In a safe state, it's possible to allocate the maximum resources requested by each process and still have enough resources available to satisfy the needs of other processes, ultimately allowing all processes to complete.
2. **Unsafe State:**
   * A system is in an unsafe state if there is no sequence of processes that can be executed without causing a deadlock.
   * In an unsafe state, allocating the maximum resources requested by each process may lead to resource exhaustion or deadlock because there are insufficient resources available to satisfy all the needs.

Example: Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: **A has 10, B has 5 and the resource type C has 7 instances.**

For find **initial instance Available = given resources – total allocation**

**Available = (10-7) (5-2) (7-5)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Allocation A         B         C** | **Max A         B         C** | **Available A         B         C** |
| P1 | 0         1          0 | 7         5         3 | 3         3         2 |
| P2 | 2         0         0 | 3         2         2 |  |
| P3 | 3         0         2 | 9         0         2 |  |
| P4 | 2         1         1 | 2         2         2 |  |
| P5 | 0         0         2 | 4         3         3 |  |
| **Total** | **7 2 5** |  |  |

**Answer the following questions using the banker's algorithm:**

1. **What is the reference of the need matrix?**
2. **Determine if the system is safe or not.**
3. **What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?**

**Solution (answer)**

***Ans. 1 : Context of the need matrix is as follows:***

**Need [i] = Max [i] - Allocation [i]**

Need for P1: (7, 5, 3) - (0, 1, 0) = **7, 4, 3**  
Need for P2: (3, 2, 2) - (2, 0, 0) = **1, 2, 2**Need for P3: (9, 0, 2) - (3, 0, 2) = **6, 0, 0**Need for P4: (2, 2, 2) - (2, 1, 1) = **0, 1, 1**Need for P5: (4, 3, 3) - (0, 0, 2) = **4, 3, 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Allocation A         B         C** | **Need A         B         C** | **Available**  (allocation + new available) | **Safe Sequence** |
| **P1** | 0         1          0 | 7         4         3 | 3 3 2 | **4th** |
| **P2** | 2         0         0 | 1         2         2 | 5 3 2 | **1st** |
| **P3** | 3         0         2 | 6         0         0 | 7 4 3 | **5th** |
| **P4** | 2         1         1 | 0         1         1 | 7 4 5 | **2nd** |
| **P5** | 0         0         2 | 4         3         1 | 7 5 5 | **3rd** |
| **Total** |  |  | **10 5 7** |  |

Hence, we created the context of need matrix.

***Ans. 2: Apply the Banker's Algorithm:***

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

**Step 1: For Process P1**

**Need <= Available**

7, 4, 3 <= 3, 3, 2 condition is **false.**

So, we examine another process, P2.

**Step 2: For Process P2**

**Need <= Available**

1, 2, 2 <= 3, 3, 2 condition is **true**

**New available = available + Allocation**

(3, 3, 2) + (2, 0, 0) => **5, 3, 2(new available)**

Similarly, we examine another process P3.

**Step 3: For Process P3**

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false.**

Similarly, we examine another process, P4.

**Step 4: For Process P4**

**P4 Need <= Available**

0, 1, 1 <= 5, 3, 2 condition **is true**

**New Available resource = Available + Allocation**

5, 3, 2 + 2, 1, 1 => **7, 4, 3(new available)**

Similarly, we examine another process P5.

**Step 5: For Process P5**

**P5 Need <= Available**

4, 3, 1 <= 7, 4, 3 condition **is true**

**New available resource = Available + Allocation**

7, 4, 3 + 0, 0, 2 => **7, 4, 5 (new available)**

Now, we **again** examine each type of resource request for ***processes P1 and P3.***

**Step 6: For Process P1**

**P1 Need <= Available**

7, 4, 3 <= 7, 4, 5 condition **is true**

**New Available Resource = Available + Allocation**

7, 4, 5 + 0, 1, 0 => **7, 5, 5(New Available)**

So, we examine another process P3.

**Step 7: For Process P3**

**P3 Need <= Available**

6, 0, 0 <= 7, 5, 5 condition **is true**

**New Available Resource = Available + Allocation**

7, 5, 5 + 3, 0, 2 => **10, 5, 7 (new available)**

Hence, we execute the banker's algorithm to find the **safe state** and the **Safe sequence like P2, P4, P5, P1 and P3.**

**Ans. 3:** For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

***For understanding Banker’s Algorithm Click below link***

[***https://youtu.be/7gMLNiEz3nw?si=vXwEuq5qwKi9U69a***](https://youtu.be/7gMLNiEz3nw?si=vXwEuq5qwKi9U69a)

* **Resource Request and Resource Release**

Resource request and resource release are fundamental concepts in operating systems that pertain to how processes or programs manage and use system resources such as CPU time, memory, I/O devices, and more. These concepts are particularly important in multi-tasking and multi-user operating systems where multiple processes run concurrently and share resources.

1. **Resource Request:**

When a process requires a particular resource to execute its tasks, it sends a resource request to the operating system. The resource request typically includes the type and quantity of the resource needed. Common examples of resource requests include:

* + **CPU Time:** A process requests CPU time to execute its instructions.
  + **Memory**: A process requests a certain amount of RAM to store its data and code.
  + **I/O Devices**: Processes may request access to input and output devices such as disk drives, network interfaces, and printers.
  + **Synchronization Primitives:** Processes often request access to synchronization mechanisms like semaphores or mutexes to coordinate with other processes.

The operating system's resource management component is responsible for handling these requests. It may grant the request if the resource is available, or it may queue the process until the resource becomes available if it's currently in use by another process.

Resource request is the action taken by a process or program to request access to a specific system resource. This resource could be a variety of things, such as memory, CPU time, files, network connections, or hardware devices.

When a process requests a resource, it typically sends a request to the operating system, which is responsible for managing and allocating resources. The operating system must check if the requested resource is available and whether granting access to it would be safe and feasible.

Resource requests are essential for programs to perform tasks, and they often include functions like **malloc()** for memory allocation, **open()** for file access, or system calls like **fork()** to create new processes.

1. **Resource Release**:

Once a process has finished using a resource or no longer requires it, it should release the resource to the operating system. This ensures that the resource can be used by other processes and prevents resource deadlock situations where processes are waiting indefinitely for resources that are held by others.

* + **CPU Time**: When a process completes its execution or voluntarily yields the CPU, it releases its hold on the CPU.
  + **Memory:** Processes release memory they've allocated when they no longer need it. This can involve deallocating memory explicitly or relying on automatic memory management (e.g., garbage collection).
  + **I/O Devices**: When a process is done with an I/O operation (e.g., reading from a file), it releases the I/O device, allowing other processes to use it.
  + **Synchronization Primitives:** Processes should release synchronization primitives when they no longer need them to allow other processes to acquire them.

The responsibility for resource release primarily rests with the processes themselves, as they are typically the best entities to determine when a resource is no longer needed. However, the operating system may also reclaim resources from processes that are terminated or no longer responding.

Resource release, on the other hand, is the action taken by a process to indicate that it has finished using a particular resource and no longer needs it. When a resource is released, it becomes available for other processes or programs to use.

The operating system must keep track of the resources allocated to processes and manage their lifecycle. When a process releases a resource, the operating system can either reclaim the resource immediately or mark it as available for future allocation, depending on the resource management strategy.

Proper resource release is crucial to prevent resource leaks and ensure that resources are efficiently utilized.