Introduction of Deadlock in Operating System

Last Updated: 30-09-2019

A process in operating systems uses different resources and uses resources in following way.  
1) Requests a resource  
2) Use the resource  
2) Releases the resource

***Deadlock***is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.  
Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/06/deadlock.png)

**Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)**  
***Mutual Exclusion:*** One or more than one resource are non-sharable (Only one process can use at a time)  
***Hold and Wait:***A process is holding at least one resource and waiting for resources.  
***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.  
***Circular Wait:*** A set of processes are waiting for each other in circular form.

**Methods for handling deadlock**  
There are three ways to handle deadlock  
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.  
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.  
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

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

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

# Deadlock Prevention

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

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

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

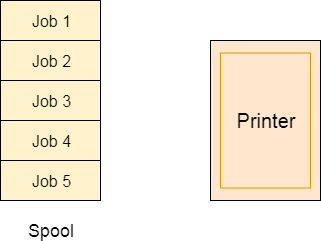
## 1. Mutual Exclusion

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

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

### Spooling

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



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

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

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

## 2. Hold and Wait

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

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

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

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

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

The problem with the approach is:

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

## 3. No Preemption

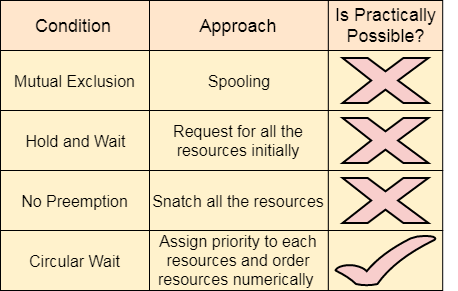
Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

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

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

## 4. Circular Wait

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



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

**Deadlock Recovery**  
A traditional operating system such as Windows doesn’t deal with deadlock recovery as it is time and space consuming process. Real-time operating systems use Deadlock recovery.

**Recovery method**

1. **Killing the process:** killing all the process involved in the deadlock. Killing process one by one. After killing each process check for deadlock again keep repeating the process till system recover from deadlock.
2. **Resource Preemption:** Resources are preempted from the processes involved in the deadlock, preempted resources are allocated to other processes so that there is a possibility of recovering the system from deadlock. In this case, the system goes into starvation.

# Deadlock avoidance

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

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

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

Safe and Unsafe States

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

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

#### Resources Assigned

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Type 1** | **Type 2** | **Type 3** | **Type 4** |
| A | 3 | 0 | 2 | 2 |
| B | 0 | 0 | 1 | 1 |
| C | 1 | 1 | 1 | 0 |
| D | 2 | 1 | 4 | 0 |

#### Resources still needed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Type 1** | **Type 2** | **Type 3** | **Type 4** |
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 1 | 2 | 1 | 0 |
| D | 2 | 1 | 1 | 2 |

1. E = (7 6 8 4)
2. P = (6 2 8 3)
3. A = (1 4 0 1)

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

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

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

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

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

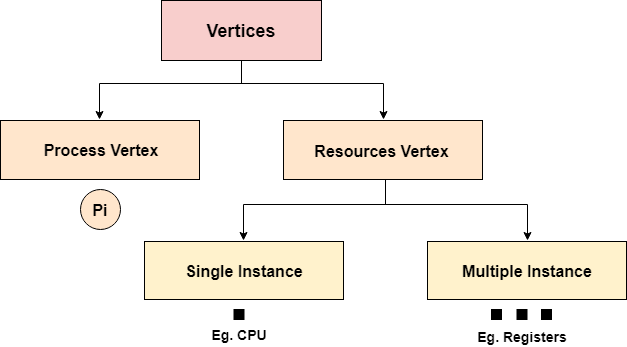
The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

# Resource Allocation Graph

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

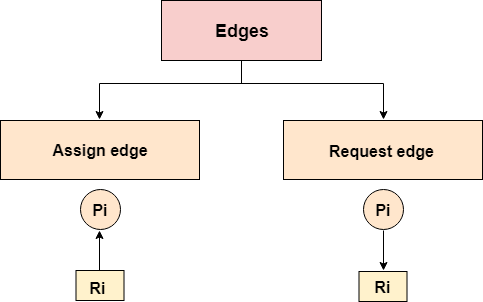
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

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



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

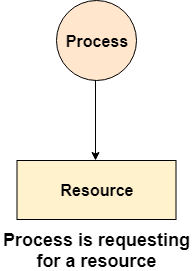
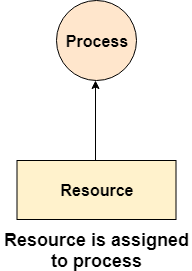
A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



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

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

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

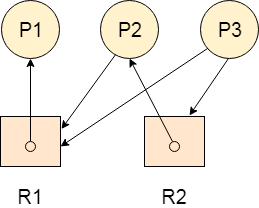
 

### Example

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

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

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



# race condition

A race condition is an undesirable situation that occurs when a [device](https://whatis.techtarget.com/definition/device) or [system](https://searchwindowsserver.techtarget.com/definition/system) attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

A simple example of a race condition is a light switch. In some homes there are multiple light switches connected to a common ceiling light. When these types of [circuits](https://whatis.techtarget.com/definition/circuit) are used, the switch position becomes irrelevant. If the light is on, moving either switch from its current position turns the light off. Similarly, if the light is off, then moving either switch from its current position turns the light on. With that in mind, imagine what might happen if two people tried to turn on the light using two different switches at exactly the same time. One [instruction](https://whatis.techtarget.com/definition/instruction) might cancel the other or the two actions might trip the circuit breaker.

Bottom of Form

Race conditions are most commonly associated with computer science. In computer [memory](https://searchstorage.techtarget.com/definition/memory-card) or [storage](https://searchstorage.techtarget.com/definition/storage), a race condition may occur if [commands](https://searchwindowsserver.techtarget.com/definition/command) to read and write a large amount of [data](https://searchdatamanagement.techtarget.com/definition/data) are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read. The result may be one or more of the following: a computer [crash](https://whatis.techtarget.com/definition/crash), an "[illegal operation](https://searchwindowsserver.techtarget.com/definition/illegal-operation)," notification and shutdown of the program, errors reading the old data or errors writing the new data. A race condition can also occur if instructions are processed in the incorrect order.

Suppose for a moment that two processes need to perform a bit [flip](https://whatis.techtarget.com/definition/flip-flops-bistable-gates) at a specific memory location. Under normal circumstances the operation should work like this:

|  |  |  |
| --- | --- | --- |
| **Process 1** | **Process 2** | **Memory Value** |
| Read value |  | 0 |
| Flip value |  | 1 |
|  | Read value | 1 |
|  | Flip value | 0 |

In this example, Process 1 performs a bit flip, changing the memory value from 0 to 1. Process 2 then performs a bit flip and changes the memory value from 1 to 0.

If a race condition occurred causing these two processes to overlap, the sequence could potentially look more like this:

|  |  |  |
| --- | --- | --- |
| **Process 1** | **Process 2** | **Memory Value** |
| Read value |  | 0 |
|  | Read value | 0 |
| Flip value |  | 1 |
|  | Flip value | 1 |

In this example, the bit has an ending value of 1 when its value should be 0. This occurs because Process 2 is unaware that Process 1 is performing a simultaneous bit flip.

### Security vulnerabilities caused by race conditions

When a program that is designed to handle tasks in a specific sequence is asked to perform two or more operations simultaneously, an attacker can take advantage of the time gap between when the service is initiated and when a security control takes effect in order to create a deadlock or thread block situation.  With deadlock, two or more threads must wait for a lock in a circular chain. This defect can cause the entire software system to halt because such locks can never be acquired if the chain is circular. Thread block can also dramatically impact application performance. In this type of concurrency defect, one thread calls a long-running operation while holding a lock and preventing the progress of other threads.

### Preventing race conditions

In computing environments, race conditions can be prevented by serialization of memory or storage access. This means if read and write commands are received close together, the read command is executed and completed first by default.

In a [network](https://searchnetworking.techtarget.com/definition/network), a race condition may occur if two users attempt to access an available [channel](https://searchdatacenter.techtarget.com/definition/channel) at the same instant, and neither computer receives notification the channel is occupied before the system grants access. Statistically, this kind of coincidence will most likely occur in networks with long lag times, such as those that use [geostationary satellites](https://searchmobilecomputing.techtarget.com/definition/geostationary-satellite). To prevent such a race condition from developing, a priority scheme must be devised. For example, the subscriber whose username begins with the earlier letter of the alphabet (or the lower numeral) may get priority by default when two subscribers attempt to access the system within a prescribed increment of time. [Hackers](https://searchsecurity.techtarget.com/definition/hacker) can take advantage of race-condition vulnerabilities to gain unauthorized access to networks.

Race conditions occasionally occur in [logic gates](https://whatis.techtarget.com/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR) when certain inputs come into conflict. Because the gate output state takes a finite, nonzero amount of time to react to any change in input states, sensitive circuits or devices following the gate may be fooled by the state of the output, and thereby not operate properly.

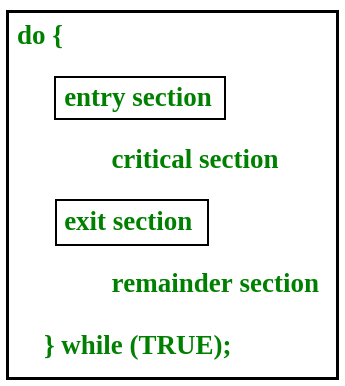
Introduction of Process Synchronization

Last Updated: 11-09-2019

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.  
   
**Race Condition**  
When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing race to say that my output is correct this condition known as  
race condition.  
Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.  
   
**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.**

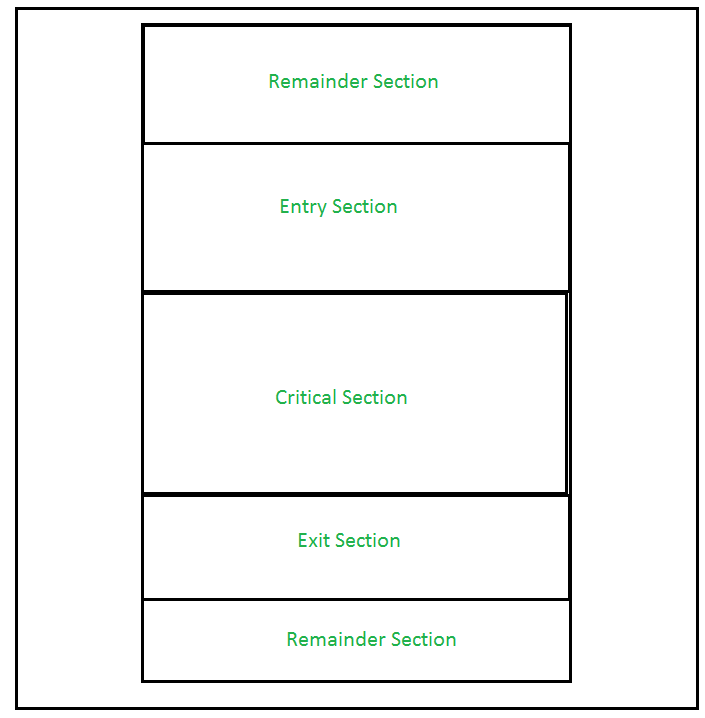
Any solution to the critical section problem must satisfy three requirements:

Mutual exclusion :  As we two process are said to follow mutual exclusion if and only if only one person are allowed to enter in critical section at one time . To check first make one process enter in the CS then try to enter other processes in the CS , if we are able to enter other processes then system are not said to follow mutual exclusion  .

Progress  : Two or more process are said to follow progress , if the decision of who will enter into critical section will not be affected by the processes who doesn’t want to enter the critical section . To check progress condition find the condition for which a process can’t enter critical section , then with condition if we are not able to enter any process into critical section then system is not said to follow progress condition otherwise yes .

Bounded Waiting : A system is said to follow bounded waiting condition if a process wants to enter into critical section will enter in some finite time .  
More understandings through Peterson solution’s

Synchronization, in which we allow only one process to enter and manipulates the shared data in Critical Section.  
//diagram of the view of CS



This setup can be defined in various regions like:

* **Entry Section –**  
  It is part of the process which decide the entry of a particular process in the Critical Section, out of many other processes.
* **Critical Section –**  
  It is the part in which only one process is allowed to enter and modify the shared variable.This part of the process ensures that only no other process can access the resource of shared data.
* **Exit Section –**  
  This process allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It checks that a process that after a process has finished execution in Critical Section can be removed through this Exit Section.
* **Remainder Section –**  
  The other parts of the Code other than Entry Section, Critical Section and Exit Section are known as Remainder Section.

# [Turn Variable | Process Synchronization](https://www.gatevidyalay.com/turn-variable-process-synchronization/)

[Operating System](https://www.gatevidyalay.com/category/subjects/operating-system/)

## ****Process Synchronization-****

Before you go through this article, make sure that you have gone through the previous article on [**Process Synchronization**](https://www.gatevidyalay.com/process-synchronization-race-condition-in-os/).

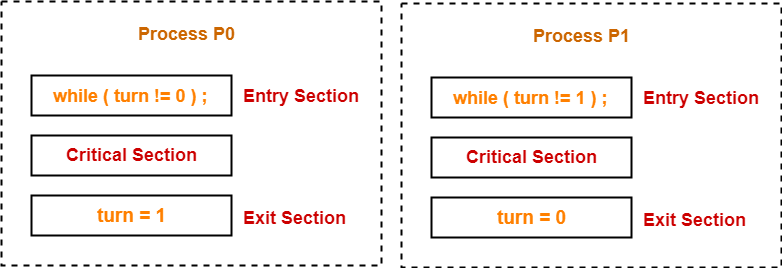
We have discussed-

* Process Synchronization provides a synchronization among the processes.
* Synchronization mechanisms allow the processes to access critical section in a synchronized manner.
* This avoids the inconsistent results.

## ****Turn Variable-****

* Turn variable is a synchronization mechanism that provides synchronization among two processes.
* It uses a turn variable to provide the synchronization.

It is implemented as-



Initially, turn value is set to 0.

* Turn value = 0 means it is the turn of process P0 to enter the critical section.
* Turn value = 1 means it is the turn of process P1 to enter the critical section.

## ****Working-****

This synchronization mechanism works as explained in the following scenes-

## ****Scene-01:****

* Process P0 arrives.
* It executes the turn!=0 instruction.
* Since turn value is set to 0, so it returns value 0 to the while loop.
* The while loop condition breaks.
* Process P0 enters the critical section and executes.
* Now, even if process P0 gets preempted in the middle, process P1 can not enter the critical section.
* Process P1 can not enter unless process P0 completes and sets the turn value to 1.

## ****Scene-02:****

* Process P1 arrives.
* It executes the turn!=1 instruction.
* Since turn value is set to 0, so it returns value 1 to the while loop.
* The returned value 1 does not break the while loop condition.
* The process P1 is trapped inside an infinite while loop.
* The while loop keeps the process P1 busy until the turn value becomes 1 and its condition breaks.

## ****Scene-03:****

* Process P0 comes out of the critical section and sets the turn value to 1.
* The while loop condition of process P1 breaks.
* Now, the process P1 waiting for the critical section enters the critical section and execute.
* Now, even if process P1 gets preempted in the middle, process P0 can not enter the critical section.
* Process P0 can not enter unless process P1 completes and sets the turn value to 0.

**Also Read-** [**Criteria For Synchronization Mechanisms**](https://www.gatevidyalay.com/critical-section-critical-section-problem/)

## ****Characteristics-****

The characteristics of this synchronization mechanism are-

* It ensures mutual exclusion.
* It follows the strict alternation approach.

|  |
| --- |
| ****Strict Alternation Approach****   In strict alternation approach,   * Processes have to compulsorily enter the critical section alternately whether they want it or not. * This is because if one process does not enter the critical section, then other process will never get a chance to execute again. |

* It does not guarantee progress since it follows strict alternation approach.
* It ensures bounded waiting since processes are executed turn wise one by one and each process is guaranteed to get a chance.
* It ensures processes does not starve for the CPU.
* It is architectural neutral since it does not require any support from the operating system.
* It is [**deadlock**](https://www.gatevidyalay.com/deadlock-in-os-conditions-for-deadlock/) free.
* It is a busy waiting solution which keeps the CPU busy when the process is actually waiting.