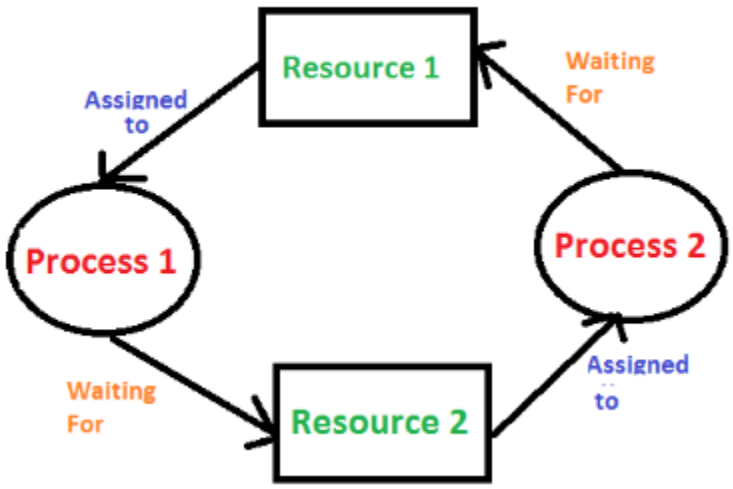
Introduction of Deadlock in Operating System

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.



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.

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

Coffman Conditions

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows:

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.

2. **Hold and Wait**

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

3. **No Preemption**

   A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.

4. **Circular Wait**

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

Deadlock Characterization

Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods:

1. All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
2. Resources can be preempted from some processes and given to others till the deadlock is resolved.

Deadlock Prevention

It is very important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight chance that a transaction may lead to deadlock in the future, it is never allowed to execute.

Deadlock Avoidance

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. The wait for graph can be used for deadlock avoidance. This is however only useful for smaller databases as it can get quite complex in larger databases.

A deadlock happens in operating system 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 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 detection, deadlock prevention and deadlock avoidance are the main methods for handling deadlocks. Details about these are given as follows:

Deadlock Detection

Deadlock can be detected by the resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be handed using the given methods:

1. All the processes that are involved in the deadlock are terminated. This approach is not that useful as all the progress made by the processes is destroyed.
2. Resources can be preempted from some processes and given to others until the deadlock situation is resolved.

Deadlock Prevention

It is important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight possibility that a transaction may lead to deadlock, it is never allowed to execute.

Some deadlock prevention schemes that use timestamps in order to make sure that a deadlock does not occur are given as follows:

- **Wait - Die Scheme**

  In the wait - die scheme, if a transaction T1 requests for a resource that is held by transaction T2, one of the following two scenarios may occur:

  - TS(T1) < TS(T2) - If T1 is older than T2 i.e T1 came in the system earlier than T2, then it is allowed to wait for the resource which will be free when T2 has completed its execution.
  - TS(T1) > TS(T2) - If T1 is younger than T2 i.e T1 came in the system after T2, then T1 is killed. It is restarted later with the same timestamp.
- **Wound - Wait Scheme**

  In the wound - wait scheme, if a transaction T1 requests for a resource that is held by transaction T2, one of the following two possibilities may occur:

  - TS(T1) < TS(T2) - If T1 is older than T2 i.e T1 came in the system earlier than T2, then it is allowed to roll back T2 or wound T2. Then T1 takes the resource and completes its execution. T2 is later restarted with the same timestamp.
  - TS(T1) > TS(T2) - If T1 is younger than T2 i.e T1 came in the system after T2, then it is allowed to wait for the resource which will be free when T2 has completed its execution.

Deadlock Avoidance

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. The wait for graph can be used for deadlock avoidance. This is however only useful for smaller databases as it can get quite complex in larger databases.

Wait for graph

The wait for graph shows the relationship between the resources and transactions. If a transaction requests a resource or if it already holds a resource, it is visible as an edge on the wait for graph. If the wait for graph contains a cycle, then there may be a deadlock in the system, otherwise not.

Ostrich Algorithm

The ostrich algorithm means that the deadlock is simply ignored and it is assumed that it will never occur. This is done because in some systems the cost of handling the deadlock is much higher than simply ignoring it as it occurs very rarely. So, it is simply assumed that the deadlock will never occur and the system is rebooted if it occurs by any chance.

Since Java 7 try-with resources was introduced. In this we declare one or more resources in the try block and these will be closed automatically after the use. (at the end of the try block)

The resources we declare in the try block should extend the java.lang.AutoCloseable class.

Example

Following program demonstrates the try-with-resources in Java.

```
import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class FileCopying {

   public static void main(String[] args) {
```

```
      try(FileInputStream inS = new FileInputStream(new
File("E:\\Test\\sample.txt"));

      FileOutputStream outS = new FileOutputStream(new
File("E:\\Test\\duplicate.txt"))){

         byte[] buffer = new byte[1024];

         int length;

         while ((length = inS.read(buffer)) > 0) {

            outS.write(buffer, 0, length);

         }

         System.out.println("File copied successfully!!");

      } catch(IOException ioe) {

         ioe.printStackTrace();

      }

   }

}
```