Deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that can be caused by another process in the set. The events that we are mainly concerned with are resource acquisition and release. The resources may be physical resources (CPU, I/O devices, memory) or logical resources (files, semaphores or monitors).

Kinds Of Resource

- **Sharable resources**:- it can be used by more than one process at a time.
- **Consumable resource**:- it can be used by only one process and the resource gets used.
- **Serially reusable resources**:- it lies between sharable and consumable resource.

Only one process can use the resource at a time but once it's done it can give it back for use by another process.

Example: CPU and Memory.

A process must request a resource before using it and must release it after using it. A resource might be Preemptable or non-preemptable.

Preemptable meaning that without harm, the resource can be borrowed form the process. Sometimes a resource can be made preemptable by the OS at some cost. Example: Memory can be preempted from a process by suspending the process and copying the contents of the memory to disk. Later the data is copied back to the memory

And the process is allowed to continue. Preemption effectively makes a serially reusable resource look sharable.

A non-preemptable resource is one that without causing undesirable efforts, cannot be taken away from the current user.

Deadlock involves non-preemptable resources.

Conditions for Deadlock

So, deadlock can arise if 4 conditions hold simultaneously:

- **Mutual exclusion**:- Only one process at a time can use a resource.
- **Hold and wait**:- A process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption**:- A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular Wait**:- Each process in a set is waiting for a resource held by the next process in the set.

Resource Allocation Graph

Deadlock is more precisely described in terms of a directed graph called Resource allocation graph. It consists of a set of all active processes and resource types in the system. Processes are described by circles and resources by squares connected by appropriate directed arrows depending on whether a resource is requested by a process or is holding on it.

- If a process is requesting a resource the arrow is directed from the process to the resource.
- If a process is holding on to a resource the arrow is directed from the resource to the process.

If a graph contains no cycles then no deadlock is present.
If a graph contains a cycle then:

- If there is only one instance per resource type then deadlock occurs.
- If there are several instances per resource type then deadlock may occur.

Methods Of Handling Deadlock

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem. Assume that deadlock rarely occurs in the system so need to address it .This is the method used by many OS including UNIX.

Deadlock Prevention

Ensure that at least one of the necessary conditions does not hold.

- Mutual Exclusion: This must hold for non-sharable resource.
- Hold and Wait: must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Requires processes to request and be allocated all its resources before it begins execution or allow resources only when the process has no resource.
  - Low resource utilization. Starvation is possible.
- No preemption: If a process that is holding some resources request another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restricted only when it can regain its old resources as well as the new ones that it is releasing.
- Circular Wait: One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

The simplest and most useful model requires that each process declares the maximum number of resources of each type that it may need. The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular wait condition. Resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Recovery from Deadlock

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?

  The following parameters need to be examined before deciding which process to terminate:

- How long the process has computed and how much longer the process will compute before completing its designated task.
- Resources the process has used.
- Resources that process needs to complete.
- How many processes will need to be terminated?
- Is the process interactive or batch?

Demonstration Of Deadlock In JAVA

The scariest thing that can happen to a java program is deadlock. It is a special type of error that you need to avoid that relates specifically to multitasking which occurs when two threads have a circular dependency on a pair of synchronized objects. This occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock so they'll sit there forever. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Deadlock is a difficult error to debug for two reasons:-

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects
  Example:-

  This example creates two classes, Dead1 and Dead2, with methods m1( ) and m2( ), respectively, which pause briefly before trying to call a method in the other class. The main class, named Deadlock, creates an Dead1 and a Dead2 instance, and then starts a second thread to set up the deadlock condition. The m1() and m2( ) methods use sleep( ) as a way to force the deadlock condition to occur.

  Code:

```
class Dead1
  {
     synchronized void m1(Dead2 b)
        {
           String name = Thread.currentThread().getName();
           System.out.println(name + " entered Dead1.m1");
           try
              {
                Thread.sleep(1000);
              } catch(Exception e)
                  {
                     System.out.println("Dead1 Interrupted");
                   }
            System.out.println(name + " trying to call Dead2.last()");
            Dead2.last();
          }

       synchronized void last()
          {
             System.out.println("Inside Dead1.last");
           }
   }

class Dead2
  {
      synchronized void m2(Dead1 a)
         {
            String name = Thread.currentThread().getName();
            System.out.println(name + " entered Dead2.m2");
            try
              {
                Thread.sleep(1000);
              } catch(Exception e)
                  {
                     System.out.println("Dead2 Interrupted");
                  }
             System.out.println(name + " trying to call Dead1.last()");
```

```
            Dead1.last();
          }

       synchronized void last()
          {
             System.out.println("Inside Dead1.last");
          }
   }

class Deadlock implements Runnable
   {
      Dead1 a = new Dead1();
      Dead2 b = new Dead2();
      Deadlock()
        {
           Thread.currentThread().setName("MainThread");
           Thread t = new Thread(this, "RacingThread");
           t.start();
           a.m1(b); // get lock on a in this thread.
           System.out.println("Back in main thread");
        }
      public void run()
        {
            b.m2(a); // get lock on b in other thread.
            System.out.println("Back in other thread");
        }
       public static void main(String args[])
        {
             new Deadlock();
        }
  }
```

On executing this program we would see the following output:

MainThread entered Dead1.m1
RacingThread entered Dead2.m2

MainThread trying to call Dead2.last()
RacingThread trying to call Dead1.last()

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC . You will see that RacingThread owns the monitor on b, while it is waiting for the monitor on a. At the same time, MainThread owns a and is waiting to get b. This program will never complete. If a multithreaded program locks up occasionally, deadlock is one of the first conditions that should be checked for.

Another Example that demonstrates deadlock:

Code:

```
public class Deadlock
   {
    private static class resource
       {
        public int value;
        }
           private Resource resourceA=new Resource();
           private Resource resourceB= new Resource();
           public int read()
              {
                 synchronized(resourceA)
                     {//May deadlock here
                         synchronized(resourceB)
                            {
                                return resourceB.value + resourceA.value;
                             }
                       }
                }

            public void write(int a ,int b)
              {
                 synchronized(resourceB)
```

```
                    {// May deadlock here
                         synchronized(resourceA)
                           {
                              resourceA.value=a;
                              resourceB.value=b;
                            }
                       }
                }
    }
```

Demonstration Of Deadlock In Database

When several transactions occur concurrently in the database the isolation property may no longer be preserved. To ensure that it is preserved the system must control the interaction among the concurrent transactions. Such mechanisms are called concurrency- control schemes. One of such scheme is based on the serializability property. One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner. That is while one transaction is accessing a data item no other transaction can modify that data item. The most common method used to implement this is to allow a transaction to access a data item only if it is currently holding a lock on that item.

LOCK

Lock is a variable associated with a data item that describes the status of the data item with respect to the possible operations that can be applied to it.
There are various modes in which a data item may be locked. But we restrict our attention to two modes:

Shared:- If a transaction T1 has obtained a lock a shared mode lock (denoted by S) on an item Q then T1 can read but cannot write Q.
Exclusive:- If a transaction T1 has obtained a lock a exclusive mode lock (denoted by X) on an item Q then T1 can both read write and Q.

It is required that every transaction request a lock in an appropriate mode on data

item Q depending on the types of operation that will be performed on Q.
But unfortunately locking can lead to an undesirable situation. Consider two transactions T1 and T2. T1 is holding an exclusive lock on a data item B. T2 is holding a shared mode lock on a data item A. Now T2 requests a shared mode lock on B. It is waiting for T1 to unlock B. T1 requests an exclusive mode lock on A. So T1 is waiting for T2 to unlock A. So a situation has arrived where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock. When deadlock occurs the system must rollback one of these two transactions. Once a transaction has been rolled back the items that were locked by that transaction are unlocked. These data items are then available to the other transaction which can then continue with its execution.

Example:

Transaction1

```
BEGIN
UPDATE ACCT_MSTR SET CURBAL=500 WHERE ACCT_NO='SB1';
UPDATE ACCT_MSTR SET CURBAL=2500 WHERE ACCT_NO='CA2'
END
```

Transaction2

```
BEGIN
UPDATE ACCT_MSTR SET CURBAL=5000 WHERE ACCT_NO='CA2';
UPDATE ACCT_MSTR SET CURBAL=3500 WHERE ACCT_NO='SB1'
END
```

When such a situation is detected by the Oracle Engine both update statements are rolled back automatically and the deadlock is resolved.

Source: http://www.go4expert.com/articles/deadlock-t22109/