**Unit 3**

**Process Deadlock**

**Deadlock:**

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. In multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources, if the resource are not available at that time, the process enters a waiting state. If the resources it has requested are held by other waiting process, then a waiting process is never again able to change state. Such situation is called deadlock.

For e.g. when two train approach each other at a crossing both shall come to a full stop and neither shall start up again until the other has gone.

Example 2: 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 as shown in figure below:

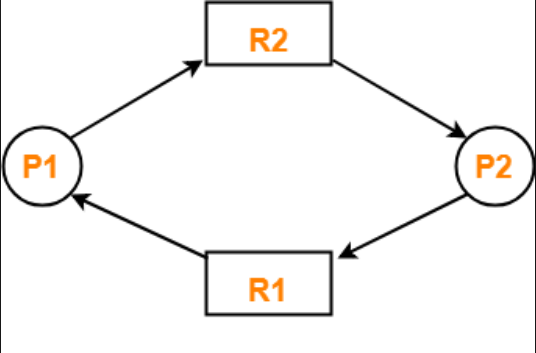


Figure: Deadlock Condition

A process must request for resource before using it and must release the resource after using it. The number of resources requested may not exceed the total number of resource available in the system. A set of process is in **deadlock** state when every process in the set is waiting for an event (resource acquisition and release) that can be caused only by another process in the set. The resource may be either physical (printer, tape drive, memory space etc.) or logical (semaphore, mutex lock and files).

**Deadlock Characterization:**

1. **Necessary Condition:**

A deadlock situation can arise if the following conditions hold simultaneously in a system:

* **Mutual exclusion:** at least one resource must be held in a non-sharable mode i.e. only one process at a time can use the resource. If another process request for such resource then the process must be delayed until the resource has been released.
* **Hold and Wait**: a process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
* **No preemption:** resource cannot be preempted i.e. a resource can be released only voluntarily by the process holding it after that process has completed its task.
* **Circular wait:** a set (P0, P1… Pn) of waiting process must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by P0.

All these condition must hold for deadlock to occur.

1. **Resource Allocation Graph:**

Deadlock can be described more precisely in terms of a directed graph called a system resource allocation graph. This graph consist of a set of vertices V and set of edges E. the set of vertices V is partitioned into two different types of node: the set containing all of the active processes P = {P1,P2 … Pn} and set containing all of the resources types in the system R = {R1, R2, … Rm}. Each process is represented as circle and each resource type as rectangle. Resource may contain more than one instance which is represented as dot in the rectangle.

If the process Pi request for the resource Rj then it is shown as Pi Rj (known as request edge).

If the resource type Rj has been allocated to Pi then it is shown as Rj Pi (known as assignment edge and in this case, edge is point to one of the instance of resource).

Following figure shows the resource allocation graph:

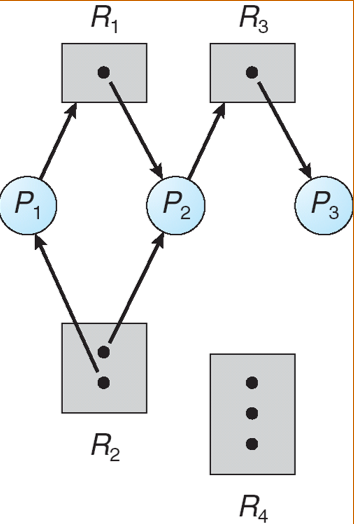


Figure: resource allocation graph

In above figure: P1 is holding an instance of R2 (R2 P2) and requesting or waiting for resource R1 (P1 R1). Process P2 is holding an instance of R2 (R2 P2), instance of R1 (R1 P2) and requesting a resource R3

(P2 R3). Process P3 is holding an instance of R3 (R3 P3).

**Condition for Deadlock:**

If the graph contains cycle then a deadlock might exist. If each resource type has exactly one instance then a cycle implies that a dead lock has occurred. Each process in this cycle is deadlocked. In this case, cycle in the graph is both necessary and a sufficient condition for the existence of deadlock.

If a resource type have several instance then a cycle does not necessarily imply that the deadlock has occurred. In this case cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Following figure shows the condition for deadlock. Here, two minimal cycle exist in the system:

* P1 R1 P2 R3 P3 R2 P1
* P2 R3 P3 R2 P2

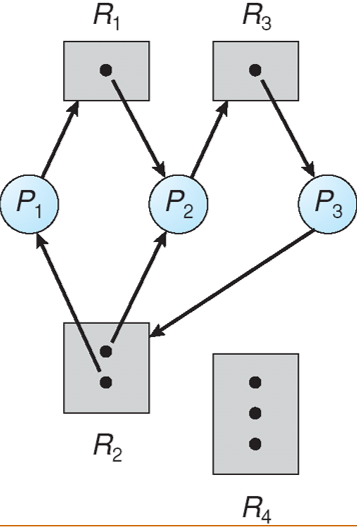


Figure: graph with deadlock

Processes P1, P2, P3 are deadlocked because P2 is waiting for the resource R3 which is already allocated to P3. Similarly, P3 is waiting for either P1 or P2 to release R2 and P1 is waiting for R2.

**Condition for no deadlock:**

If the graph does not contains any cycle the process is not deadlock. Even if there is cycle, there might not be a chance of deadlock if the resource contains multiple instances. Following figure shows the graph with cycle but no deadlock.

Here, cycle exist in the system but does not contains any deadlock:

* P1 R1 P3 R2 P1

P4 may release its instance of resource type R2. Such resource can be allocated to P3 breaking the cycle.

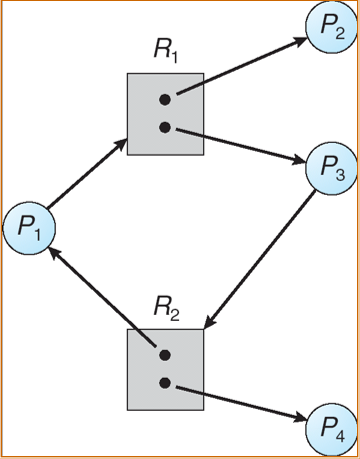


Figure: Graph with a cycle but no deadlock.

**Methods for Handling Deadlocks:**

Deadlock problem can be handle in one of the three ways:

* Protocols can be used to prevent or avoid deadlock ensuring that system will never enter a deadlock state
* Allowing the system to enter a deadlock state, detect it and recover
* Ignoring the problem and pretending that deadlock never occur in the system.

1. **Deadlock Prevention:**

It provides a set of method to ensure that at least one of the necessary conditions (mutual exclusion, hold and wait, no preemption and circular wait) cannot hold. By ensuring that at least one of this condition cannot hold, we can prevent the occurrence of deadlock.

1. **Mutual exclusion:**

This condition must hold i.e. at least one resource must be non- sharable. Sharable resource do not require mutually exclusive access and thus cannot be involve in a deadlock. A process never need to wait for sharable resources.

If we can able to violate resource behaving in mutually exclusive manner then we can prevent the dead lock. For this spooling can be used in which separate memory is associated which stores the job from each of the process into it. By using this mechanism process does not have to wait for job and can continue whatever it is doing.

1. **Hold and Wait:**

To ensure the hold and wait condition never occur in the system, it must be guarantee that whenever a process request a resource it does not hold any other resource. Two types of protocol are used for this purpose.

* One protocol requires each process to request and be allocated all its resources before it begins execution. This provision can be implement by requiring that system calls requesting resources for a process precede all other system calls.
* An alternative protocol allows a process to request resource only when it has none. A process may request resource and use them. Before it can request any additional resource it should release all the resources that it is acquiring or holding.

To illustrate these case let us consider an example in which a process need to copy data from DVD drive to the disk and print the result to a printer.

If used first protocol then process initially request all the resource i.e. DVD, disk and printer. It will hold the printer for its entire execution even though it needs the printer only at the end.

If used second protocol then it allows a process to request initially only the DVD drive and disk to copies the data and release both DVD and disk. The process then request disk and printer to copy the file to printer. After the work is done process must releases both resource and terminates.

**Disadvantages**:

* Resource utilization may be low as resource are unused for long period of time although resources are allocated.
* Starvation is possible. A process that needs several other resources have to wait because at least one of the resources that it need is always allocated to some other process.

1. **No preemption:**

The third condition for possibility of deadlock is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold following protocol can be used:

* If a process is holding some resource and requests another resource that cannot be immediately allocated then all the resource that a process is currently holding are preempted. The process will be restarted when it can regain its old resources as well as the new one that it was request.
* Alternately, if a process request some resources first the availability of such resource is check. If they are not available, then search is done to ensure that whether that resource are allocated to other process that is waiting for another resource. If so then the desire resource is preempted from such process and allocated to requesting process.

1. **Circular Wait:**

One way to ensure that this condition does not hold is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration. To illustrate this let us suppose R= {R1, R2 … Rn} be the set of resource types. Unique integer number is assign to each resource in order to compare two resource and to determine whether one process precedes another or not. For e.g. if the resource set contains three resource: tape drive, disk drive and printer then function might be define as:

F (tape drive) = 1

F (disk drive) = 5

F (printer) = 12

In order to prevent deadlock, each process can request resource only in an increasing order of enumeration. The process can initially request any instance of a resource let say Ri. After that, the process can request instances of other resource let say Rj if and only if F (Rj) > = F (Ri). For example if a process wants tape drive and disk drive at the same time then first tape drive have to be requested and then disk drive should be request.

Alternately, it can be require that the process requesting an instance of resource type Rj must have release any resource type Ri such that F(Ri) > =F(Rj). For e.g. if a process is holding resource printer and if it requires disk driver then in this case F (printer) > F (disk drive) so a process have to release the resource disk drive.

1. **Deadlock Avoidance:**

An alternative method for avoiding deadlock is to require additional information about how resources are to be requested. Each request requires that the system consider the resources currently available, the resource currently allocated to each process and the future request and releases of each process.

The various algorithm that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resource of each type that it may need. Given this information, algorithm can be construct that ensures that the system will never enter a deadlock state.

A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist. The resource allocation state is defined by the number of available and allocated resource and the maximum demand of the process.

Following are the deadlock avoidance algorithm:

1. **Resource Allocation graph Algorithm:**

In addition to the request and assignment edges in resource allocation graph here a new type of edge is used known as claim edge. A claim edge Pi Rj indicates that process Pi may request resource Rj at some point of time. Claim edge is represented by dashed line in graph. When a process Pi request resource Rj then claim edge is converted to request edge (dashed line is converted to straight line). When resource Rj is released by Pi then the assignment edge Rj Pi is reconverted to claim edge Pi Rj.

When the process Pi requests resource Rj then the request is granted only if converting the request edge Pi Rj to an assignment edge Rj Pi does not forms the cycle. If no cycle is exist then the allocation of the resource will leave the system in a safe state. If a cycle is found then the allocation will put the system in an unsafe state. In this case, Pi have to wait for its request to be satisfied.

Following figure shows the resource allocation graph for deadlock avoidance:

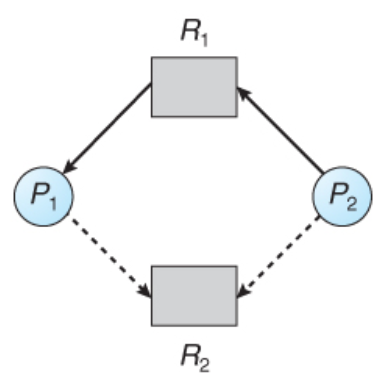


Figure: Resource allocation graph for deadlock avoidance.

To illustrate this algorithm let us consider that P2 request R2. Although R2 is currently free, it cannot be allocated because this will create a cycle in a graph i.e. system is in unsafe state.

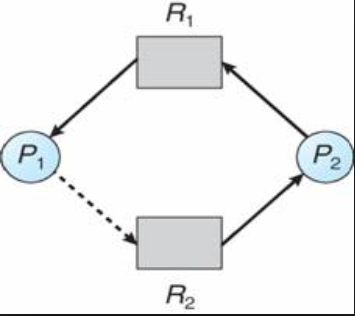


Figure: An unsafe state in a resource allocation graph

1. **Banker’s Algorithm:**

Based on the safety algorithm this algorithm is applicable to a resource with multiple instance and use to find out whether the system is in safe state or not and whether the request can be safely granted or not. It uses two algorithm: safety algorithm and resource request algorithm.

When a new process enters the system, it must declare the maximum number of instance of each resource type that it may need. When the set of resources are requested then the system must determine whether the allocation of these resource will leave the system in safe state or not. If it will then the resource are allocated, if not then the process have to wait until other process release enough resources.

Data structure:

Here “n” is the number of process in the system and m is the number of resource type.

* **Available:** a vector of length m indicates the number of available resources of each type. If Available [j] equals k, then k instance of resource type Rj are available.
* **Max:** An n \* m matrix defines the maximum demand of each process. If Max [i] [j] equals k then process Pi may request at most k instance of resource type Rj.
* **Allocation:** an n\*m matrix defines the number of resources of each type currently allocated to each process. If Allocation [i][j] equals k then process Pi is currently allocated k instances of resource type Rj.
* **Need:** an n \* m matrix indicates the remaining resource need of each type. If Need [i] [j] equals k then process Pi may need k more instance of resource type Rj to complete its task. Mathematically: Need[i] [j] = Max [i] [j] - Allocation [i][j]

1. **Safety algorithm:**

This algorithm finds out whether the system is in safe state or not. This algorithm can be described as:

* Step 1: Need matrix = max – allocation
* Step 2: if (need < = available){

Execute process;

New available = available + allocation

}

Else{

Do not execute go forward

}

1. **Resource Request Algorithm:**

It determines whether requests can be safely granted or not.

* Step 1: if request < = need then go to step 2

Else error

* Step 2: if request < = available, go to step 3

Else wait

* Step 3:

Available = available – request

Allocation = allocation + request

Need = need – request

Example on Bankers Algorithm

**Question 1**

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Allocation | Max | Available |
| P0  P1  P2  P3  P4 | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 1 | 2 | | 1 | 0 | 0 | 0 | | 1 | 3 | 5 | 4 | | 0 | 6 | 3 | 2 | | 0 | 0 | 1 | 4 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 1 | 2 | | 1 | 7 | 5 | 0 | | 2 | 3 | 5 | 6 | | 0 | 6 | 5 | 2 | | 0 | 6 | 5 | 6 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 1 | 5 | 2 | 0 | |

* Find the need matrix?
* Is the system is in safe state? If yes find the safe sequence?
* If P1 request (0, 4, 2, 0). Can the request be granted immediately?

**Solution:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Allocation | Max | Available | Need |
| P0  P1  P2  P3  P4 | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 1 | 2 | | 1 | 0 | 0 | 0 | | 1 | 3 | 5 | 4 | | 0 | 6 | 3 | 2 | | 0 | 0 | 1 | 4 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 1 | 2 | | 1 | 7 | 5 | 0 | | 2 | 3 | 5 | 6 | | 0 | 6 | 5 | 2 | | 0 | 6 | 5 | 6 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 1 | 5 | 2 | 0 | | 1 | 5 | 3 | 2 | | 2 | 8 | 8 | 6 | | 2 | 14 | 11 | 8 | | 2 | 14 | 12 | 12 | | 3 | 14 | 12 | 12 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 0 | 0 | | 0 | 7 | 5 | 0 | | 1 | 0 | 0 | 2 | | 0 | 0 | 2 | 0 | | 0 | 6 | 4 | 2 | |

**Safe Sequence: P0 => P2 => P3 => P4 => P1**

**For need matrix:**

Need = max – allocation

For P0 :

Need = max of P0 - allocation of P0

= 0 0 1 2 – 0 0 1 2

= 0 0 0 0

Similarly do for other process (p1 to P4) and insert the result in need column

Step 2:

**At first**

Since for p0: need(0 0 0 0 ) <= available(1 5 2 0) P0 is executed first.

New available = old available +allocation

New available = 1 5 2 0 + 0 0 1 2 = 1 5 3 2

**At second**

Since for P2: need(1 0 0 2) <= available (1 5 3 2) so p2 is execute at second)

New available = old available + allocation

New available = 1 5 3 2 + 1 3 5 4

New available = 2 8 8 6

**At third:**

Since for P3: need (0 0 2 0)< = available (2 8 8 6) P3 is executed

New available = old available +allocation

New available = 2 8 8 6+0 6 3 2

New available = 2 14 11 8

**At fourth:**

Since for P4: need (0 6 4 2 ) < = available (2 14 11 8) P4 is executed at fourth

New available = old available + allocation

New available = 2 14 11 8 + 0 0 1 4

New available = 2 14 12 12

**At last:**

Since P1: need (0 7 5 0) <= available (2 14 12 12) P1 is executed

New available = 2 14 12 12 + 1 0 0 0

New available = 3 14 12 12

Since all the process are executed, system is in safe state. Safe sequence is P0 => P2=> P3=> P4=> P1

**If P1 request (0, 4, 2, 0). Can the request be granted immediately?**

**P1 requested 0 4 2 0:**

Request of P1 = 0 4 2 0

Need of P1= 0 7 5 0

Available = 1 5 2 0

**Step 1**: request <=need

0 4 2 0 < = 0 7 5 0 true

**(Note: If in case step 1 will be false write resource cannot be granted)**

**Step 2**: request < = available

0 4 2 0 < = 1 5 2 0 true

**(note: if step 1 is true but step 2 is false then write resource cannot be granted)**

**Step 3:**

New available = available – request

New available = 1 5 2 0 – 0 4 2 0

**New available** = 1 1 0 0

Allocation = old allocation + request

Allocation = 1 0 0 0 + 0 4 2 0

**Allocation** = 1 4 2 0

Need = need – request

Need = 0 7 5 0 – 0 4 2 0

**Need** = 0 3 3 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Allocation | Max | Available | Need |
| P0  P1  P2  P3  P4 | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 1 | 2 | | 1 | 4 | 2 | 0 | | 1 | 3 | 5 | 4 | | 0 | 6 | 3 | 2 | | 0 | 0 | 1 | 4 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 1 | 2 | | 1 | 7 | 5 | 0 | | 2 | 3 | 5 | 6 | | 0 | 6 | 5 | 2 | | 0 | 6 | 5 | 6 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 1 | 1 | 0 | 0 | | 1 | 1 | 1 | 2 | | 2 | 4 | 6 | 6 | | 2 | 10 | 9 | 8 | | 2 | 10 | 10 | 12 | | 3 | 14 | 12 | 12 | | |  |  |  |  | | --- | --- | --- | --- | | A | B | C | D | | 0 | 0 | 0 | 0 | | 0 | 3 | 3 | 0 | | 1 | 0 | 0 | 2 | | 0 | 0 | 2 | 0 | | 0 | 6 | 4 | 2 | |

After this find out safe sequence using safety algorithm (same as first one). Start from P0.

**For P0:**

need (0 0 0 0 ) < = available ( 1 1 0 0 ) so p0 is executed first

New available = available + allocation

= 1 1 0 0 + 0 0 1 2

= 1 1 1 2

**For P2:**

need (1 0 0 2 ) < = available ( 1 1 1 2 ) so p2 is executed second

New available = available +allocation

= 1 1 1 2 + 1 3 5 4

= 2 4 6 6

**For P3:**

need (0 0 2 0 ) < = available ( 2 4 6 6 ) so p3 is executed third

New available = available +allocation

= 2 4 6 6 + 1 3 5 4

= 2 4 6 6 + 0 6 3 2 = 2 10 9 8

**For P4:**

need (0 6 4 2 ) < = available ( 2 10 9 8 ) so p4 is executed fourth

New available = available +allocation

= 2 10 9 8 + 0 0 1 4

= 2 10 10 12

**For P1:**

need (0 3 3 0 ) < = available ( 2 10 10 12 ) so p1 is executed last

New available = available +allocation

= 2 10 10 12 + 1 4 2 0

= 3 14 12 12

Safe Sequence: **P0 => P2 => P3 => P4 => P1**

**if the processes are executed in P0 => P2 => P3 => P4 => P1** **sequence system will be on safe state**

**Recovery from the Deadlock:**

When it is proved that deadlock exist in system then several alternatives are available. There are two option for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlock processes.

1. **Process Termination:**

Two methods are used to eliminate the deadlock by aborting a process.

* Abort all deadlock process:

This method will break the deadlock cycle, but at great expense. The deadlock processes may have computed for a long time and the results of these partial computations must be discarded and probably will have to be recomputed later.

* Abort one process at a time until the deadlock cycle is eliminated:

After each process is aborted, a deadlock detection algorithm must be invoked to determine whether any process are still in deadlock or not. So, it contains considerable overhead.

**Disadvantage:**

* If the process was is the middle of updating file, terminating it will leave that file in an incorrect state. Here, deadlock process that should be terminated have to determine in advance.

1. **Resource preemption:**

Using this method, some resource are preempted from process and such resource are given to another process until the deadlock cycle is broken. To use this method, following three issues should be addressed:

* **Selecting a victim:** which resource and which processes are to be preempted? The order of the preemption should be determine such that it minimize the cost.
* **Rollback:** if the resource is preempted from process, what should be done with that process? It cannot continue normal execution because resource have been taken from it. So, the process should be roll back to safe state and restart from that state.
* **Starvation:** how can we guarantee that resource will not be preempted from the same process? We must ensure that a process can be picked as a victim only for finite number of time. The most common solution is to include the number of rollback in the cost factor.