MODULE 3: DEADLOCKS
MEMORY MANAGEMENT

MODULE 3: DEADLOCKS

3.1 Deadlocks

• Deadlock is a situation where a set of processes are blocked because each process is
→ holding a resource and
→ waiting for another resource held by some other process.

• Real life example:
When 2 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).

• Here is an example of a situation where deadlock can occur (Figure 3.1).

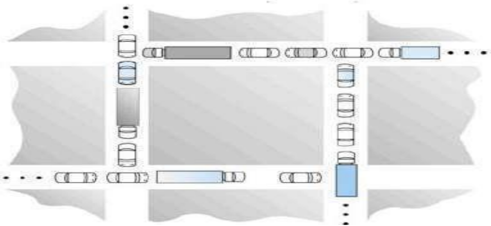


Figure 3.1 Deadlock Situation

3.2 System Model

• A system consist of finite number of resources. (For ex: memory, printers, CPUs).

• These resources are distributed among number of processes.

• A process must
→ request a resource before using it and
→ release the resource after using it.

• The process can request any number of resources to carry out a given task.

• The total number of resource requested must not exceed the total number of resources available.

• In normal operation, a process must perform following tasks in sequence:

1) Request
➢ If the request cannot be granted immediately (for ex: the resource is being used by another process), then the requesting-process must wait for acquiring the resource.
➢ For example: open( ), malloc( ), new( ), and request( )

2) Use
➢ The process uses the resource.
➢ For example: prints to the printer or reads from the file.

3) Release
➢ The process releases the resource.
➢ So that, the resource becomes available for other processes.
➢ For example: close( ), free( ), delete( ), and release( ).

• A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set.

• Deadlock may involve different types of resources.

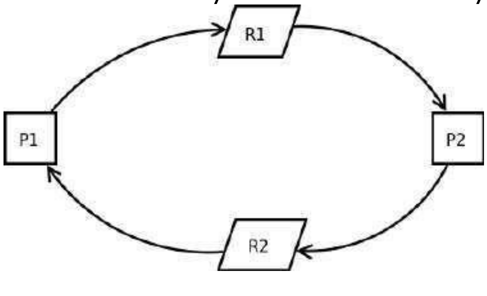


Figure 3.2

• As shown in figure 3.2,
Both processes P1 & P2 need resources to continue execution.
P1 requires additional resource R1 and is in possession of resource R2.
P2 requires additional resource R2 and is in possession of R1.

• Thus, neither process can continue.

• Multithread programs are good candidates for deadlock because they compete for shared resources.

3.3 Deadlock Characterization

• In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

3.3.1 Necessary Conditions

• There are four conditions that are necessary to achieve deadlock:

1) Mutual Exclusion

➢ At least one resource must be held in a non-sharable mode.

➢ If any other process requests this resource, then the requesting-process must wait for the resource to be released.

2) Hold and Wait

➢ A process must be simultaneously

→ holding at least one resource and

→ waiting to acquire additional resources held by the other process.

3) No Preemption

➢ Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

4) Circular Wait

➢ A set of processes { P0, P1, P2, . . ., PN } must exist such that

P0 is waiting for a resource that is held by P1

P1 is waiting for a resource that is held by P2, and so on

3.3.2 Resource-Allocation-Graph

• The resource-allocation-graph (RAG) is a directed graph that can be used to describe the deadlock situation.

• RAG consists of a
  → set of vertices (V) and
  → set of edges (E).

• V is divided into two types of nodes
  1) P={P1,P2…....Pn} i.e., set consisting of all active processes in the system.
  2) R={R1,R2…......Rn} i.e., set consisting of all resource types in the system.

• E is divided into two types of edges:

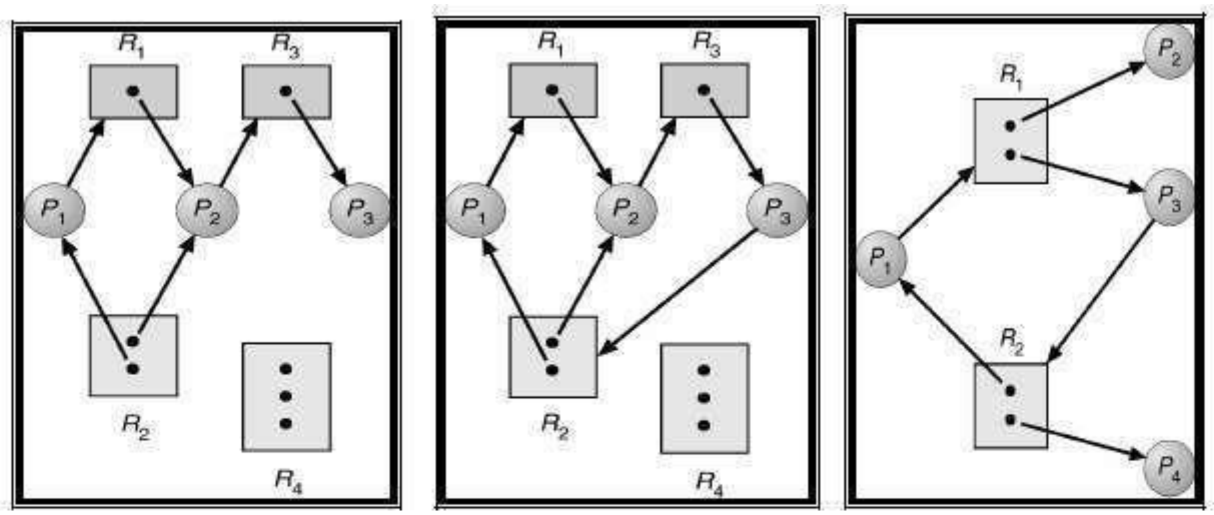
  1) Request Edge
  ➢ A directed-edge $P_i \rightarrow R_j$ is called a request edge.
  ➢ $P_i \rightarrow R_j$ indicates that process P_i has requested a resource R_j.

  2) Assignment Edge
  ➢ A directed-edge $R_j \rightarrow P_i$ is called an assignment edge.
  ➢ $R_j \rightarrow P_i$ indicates that a resource R_j has been allocated to process P_i.

• Suppose that process Pi requests resource Rj.
  Here, the request for Rj from Pi can be granted only if the converting request-edge to assignment-edge do not form a cycle in the resource-allocation graph.

• Pictorially,
  → We represent each process P_i as a **circle**.
  → We represent each resource-type R_j as a **rectangle**.

• As shown in below figures, the RAG illustrates the following 3 situation (Figure 3.3):
  1) RAG with a deadlock
  2) RAG with a cycle and deadlock
  3) RAG with a cycle but no deadlock



(a) Resource allocation Graph (b) With a deadlock (c) with cycle but no deadlock
Figure 3.3 Resource allocation graphs

Conclusion:
  1) If a graph contains no cycles, then the system is not deadlocked.
  2) If the graph contains a cycle then a deadlock may exist.
  Therefore, a cycle means deadlock is possible, but not necessarily present.

3.4 Methods for Handling Deadlocks

• There are three ways of handling deadlocks:

1) Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
2) Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
3) Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot the system.

• In order to avoid deadlocks, the system must have additional information about all processes.
• In particular, the system must know what resources a process will or may request in the future.
• Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources.
• If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down.

3.5 Deadlock-Prevention

• Deadlocks can be eliminated by preventing at least one of the four required conditions:

1) Mutual exclusion
2) Hold-and-wait
3) No preemption
4) Circular-wait.

3.5.1 Mutual Exclusion

• This condition must hold for non-sharable resources.
• For example:

A printer cannot be simultaneously shared by several processes.

• On the other hand, shared resources do not lead to deadlocks.
• For example:

Simultaneous access can be granted for read-only file.

• A process never waits for accessing a sharable resource.
• In general, we cannot prevent deadlocks by denying the mutual-exclusion condition because some resources are non-sharable by default.

3.5.2 Hold and Wait

• To prevent this condition:

The processes must be prevented from holding one or more resources while simultaneously waiting for one or more other resources.

• There are several solutions to this problem.
• For example:

Consider a process that

→ copies the data from a tape drive to the disk
→ sorts the file and
→ then prints the results to a printer.

Protocol-1

➢ Each process must be allocated with all of its resources before it begins execution.
➢ All the resources (tape drive, disk files and printer) are allocated to the process at the beginning.

Protocol-2

➢ A process must request a resource only when the process has none.
➢ Initially, the process is allocated with tape drive and disk file.
➢ The process performs the required operation and releases both tape drive and disk file.
➢ Then, the process is again allocated with disk file and the printer
➢ Again, the process performs the required operation & releases both disk file and the printer.

• Disadvantages of above 2 methods:

1) Resource utilization may be low, since resources may be allocated but unused for a long period.
2) Starvation is possible.

3.5.3 No Preemption

• To prevent this condition: the resources must be preempted.
• There are several solutions to this problem.

Protocol-1

• If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted.
• The preempted resources are added to the list of resources for which the process is waiting.
• The process will be restarted only when it regains the old resources and the new resources that it is requesting.

Protocol-2

• When a process request resources, we check whether they are available or not.

```
If (resources are available)
then
{
        allocate resources to the process
}
else
{
        If (resources are allocated to waiting process)
        then
        {
                preempt the resources from the waiting process
                allocate the resources to the requesting-process
                the requesting-process must wait
        }

}
```

• These 2 protocols may be applicable for resources whose states are easily saved and restored, such as registers and memory.
• But, these 2 protocols are generally not applicable to other devices such as printers and tape drives.

3.5.4 Circular-Wait

• Deadlock can be prevented by using the following 2 protocol:

Protocol-1

➢ Assign numbers all resources.
➢ Require the processes to request resources only in increasing/decreasing order.

Protocol-2

➢ Require that whenever a process requests a resource, it has released resources with a lower number.

• One big challenge in this scheme is determining the relative ordering of the different resources.

3.6 Deadlock Avoidance

• The general idea behind deadlock avoidance is to prevent deadlocks from ever happening.
• Deadlock-avoidance algorithm
→ requires more information about each process, and
→ tends to lead to low device utilization.
• For example:
1) In simple algorithms, the scheduler only needs to know the maximum number of each resource that a process might potentially use.
2) In complex algorithms, the scheduler can also take advantage of the schedule of exactly what resources may be needed in what order.
• A deadlock-avoidance algorithm dynamically examines the resources allocation state to ensure that a circular-wait condition never exists.
• The resource-allocation state is defined by
→ the number of available and allocated resources and
→ the maximum demand of each process.

3.6.1 Safe State

• A state is safe if the system can allocate all resources requested by all processes without entering a deadlock state.
• A state is safe if there exists a safe sequence of processes {P0, P1, P2, ..., PN} such that the requests of each process(Pi) can be satisfied by the currently available resources.
• If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock.
• All safe states are deadlock free, but not all unsafe states lead to deadlocks. (Figure 3.4).

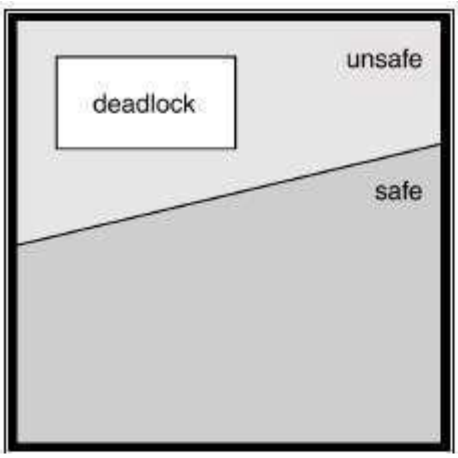


Figure 3.4 Safe, unsafe, and deadlock state spaces

3.6.2 Resource-Allocation-Graph Algorithm

• If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
• In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges (denoted by a dashed line).
• Claim edge Pi → Rj indicated that process Pi may request resource Rj at some time in future.
• The important steps are as below:

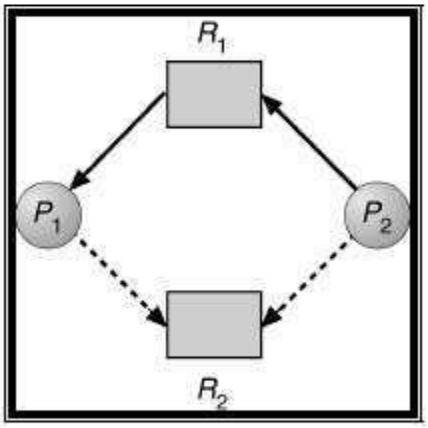
  1) When a process Pi requests a resource Rj, the claim edge Pi → Rj is converted to a request edge.
  2) Similarly, when a resource Rj is released by the process Pi, the assignment edge Rj → Pi is reconverted as claim edge Pi → Rj.
  3) The request for Rj from Pi can be granted only if the converting request edge to assignment edge do not form a cycle in the resource allocation graph.

• To apply this algorithm, each process Pi must know all its claims before it starts executing.
• Conclusion:

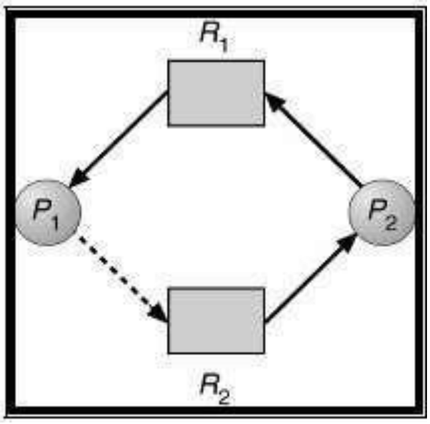
  1) If no cycle exists, then the allocation of the resource will leave the system in a safe state.
  2) If cycle is found, system is put into unsafe state and may cause a deadlock.

• For example: Consider a resource allocation graph shown in Figure 3.5(a).

  ➢ Suppose P2 requests R2.
  ➢ Though R2 is currently free, we cannot allocate it to P2 as this action will create a cycle in the graph as shown in Figure 3.5(b).
  ➢ This cycle will indicate that the system is in unsafe state: because, if P1 requests R2 and P2 requests R1 later, a deadlock will occur.



(a) For deadlock avoidance



(b) an unsafe state

Figure 3.5 Resource Allocation graphs

• Problem:

  The resource-allocation graph algorithm is not applicable when there are multiple instances for each resource.

• Solution:

  Use banker's algorithm.

3.6.3 Banker's Algorithm

- This algorithm is applicable to the system with multiple instances of each resource types.
- However, this algorithm is less efficient then the resource-allocation-graph algorithm.
- When a process starts up, it must declare the maximum number of resources that it may need.
- This number may not exceed the total number of resources in the system.
- When a request is made, the system determines whether granting the request would leave the system in a safe state.
- If the system in a safe state,
  the resources are allocated;
  else
  the process must wait until some other process releases enough resources.
- Assumptions:
  Let n = number of processes in the system
  Let m = number of resources types.
- Following data structures are used to implement the banker's algorithm.
  1) Available [m]
  - This vector indicates the no. of available resources of each type.
  - If Available[j]=k, then k instances of resource type Rj is available.

  2) Max [n][m]
  - This matrix indicates the maximum demand of each process of each resource.
  - If Max[i,j]=k, then process Pi may request at most k instances of resource type Rj.

  3) Allocation [n][m]
  - This matrix indicates no. of resources currently allocated to each process.
  - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.

  4) Need [n][m]
  - This matrix indicates the remaining resources need of each process.
  - If Need[i,j]=k, then Pi may need k more instances of resource Rj to complete its task.
  - So, Need[i,j] = Max[i,j] - Allocation[i]
- The Banker's algorithm has two parts: 1) Safety Algorithm
  2) Resource – Request Algorithm

3.6.3.1 Safety Algorithm

- This algorithm is used for finding out whether a system is in safe state or not.
- Assumptions:
  Work is a working copy of the available resources, which will be modified during the analysis.
  Finish is a vector of boolean values indicating whether a particular process can finish.

Step 1:
Let Work and Finish be two vectors of length m and n respectively.
Initialize:
Work = Available
Finish[i] = false for i=1,2,3,…….n

Step 2:
Find an index(i) such that both
a) Finish[i] = false
b) Need i <= Work.
If no such i exist, then go to step 4

Step 3:
Set:
Work = Work + Allocation(i)
Finish[i] = true
Go to step 2

Step 4:
If Finish[i] = true for all i, then the system is in safe state.

3.6.3.2 Resource-Request Algorithm

• This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
• When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request.
• Let Request(i) be the request vector of process Pi.
• If Request(i)[j]=k, then process Pi wants K instances of the resource type Rj.

```
Step 1:
        If Request(i) <= Need(i)
        then
                go to step 2
        else
                raise an error condition, since the process has exceeded its maximum claim.
Step 2:
        If Request(i) <= Available
        then
                go to step 3
        else
                Pi must wait, since the resources are not available.
Step 3:
        If the system want to allocate the requested resources to process Pi then modify the
        state as follows:
                Available = Available – Request(i)
                Allocation(i) = Allocation(i) + Request(i)
                Need(i) = Need(i) – Request(i)
Step 4:
        If the resulting resource-allocation state is safe,
                then    i) transaction is complete and
                        ii) Pi is allocated its resources.
Step 5:
         If the new state is unsafe,
                 then   i) Pi must wait for Request(i) and
                        ii) old resource-allocation state is restored.
```

3.6.3.3 An Illustrative Example

Question: Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	3	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Answer the following questions using Banker's algorithm.

i) What is the content of the matrix need?
ii) Is the system in a safe state?
iii) If a request from process P1 arrives for (1 0 2) can the request be granted immediately?

Solution (i):

• The content of the matrix Need is given by
  Need = Max - Allocation
• So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Solution (ii):

• Applying the Safety algorithm on the given system,

Step 1: Initialization
  Work = Available i.e. Work =3 3 2
  ……P0………P1……..P2……..P3……P4…..
  Finish = | false | false | false | false | false |

Step 2: For i=0
  Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(3 3 2) → false
  So P0 must wait.

Step 2: For i=1
  Finish[P1] = false and Need[P1]<=Work i.e. (1 2 2)<=(3 3 2) → true
  So P1 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P1] =(3 3 2)+(2 0 0)=(5 3 2)
  …….P0………P1…….P2…….P3… ... P4……
  Finish = | false | true | false | false | false |

Step 2: For i=2
  Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0)<=(5 3 2) → false
  So P2 must wait.

Step 2: For i=3
  Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) → true
  So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3)
  ……P0………P1……P2………..P3…….P4….
  Finish = | false | true | false | true | false |

Step 2: For i=4
Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) → true
So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5)
……P0………P1……P2………P3…….P4…..
Finish= | false | true | false | true | true |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) → true
So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5)
…..P0…….P1………P2…….P3…….P4….
Finish= | true | true | false | true | true |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) → true
So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7)
…..P0…….P1………P2…….P3…….P4….
Finish= | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is currently in a safe state.
The safe sequence is <P1, P3, P4, P0, P2>.

Conclusion: Yes, the system is currently in a safe state.

Solution (iii): P1 requests (1 0 2) i.e. Request[P1]=1 0 2

- To decide whether the request is granted, we use Resource Request algorithm.

Step 1: Request[P1]<=Need[P1] i.e. (1 0 2)<=(1 2 2) → true.
Step 2: Request[P1]<=Available i.e. (1 0 2)<=(3 3 2) → true.
Step 3: Available = Available – Request[P1] = (3 3 2) - (1 0 2)= (2 3 0)
Allocation[P1] = Allocation[P1] + Request[P1] = (2 0 0) + (1 0 2)= (3 0 2)
Need[P1] = Need[P1] – Request[P1] = (1 2 2) - (1 0 2)= (0 2 0)

- We arrive at the following new system state:

	Allocation			**Max**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	2	3	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

- The content of the matrix Need is given by
  Need = Max - Allocation
- So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

- To determine whether this new system state is safe, we again execute Safety algorithm.

Step 1: Initialization
Here, m=3, n=5
Work = Available i.e. Work =2 3 0
…..P0………P1………P2……….P3…….P4….
Finish = | false | false | false | false | false |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(2 3 0) → false
So P0 must wait.

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (0 2 0)<=(2 3 0) → true
So P1 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P1] =(2 3 0)+(3 0 2)=(5 3 2)
……P0………P1…….P2………P3…….P4……
Finish = | false | true | false | false | false |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(5 3 2) → false
So P2 must wait.

Step 2: For i=3
Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) → true
So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3)
…..P0……….P1……P2……..P3…….P4…...
Finish = | false | true | false | true | false |

Step 2: For i=4
Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) → true
So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5)
……P0………P1……P2………P3…….P4….
Finish = | false | true | false | true | true |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) → true
So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5)
……P0…….P1………P2……P3 ..... P4….
Finish = | true | true | false | true | true |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) → true
So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7)
…..P0…….P1………P2…….P3…….P4….
Finish= | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is in a safe state.
The safe sequence is <P1, P3, P4, P0, P2>.

Conclusion: Since the system is in safe sate, the request can be granted.

3.7 Deadlock Detection

- If a system does not use either deadlock-prevention or deadlock-avoidance algorithm then a deadlock may occur.
- In this environment, the system must provide
  1) An algorithm to examine the system-state to determine whether a deadlock has occurred.
  2) An algorithm to recover from the deadlock.

3.7.1 Single Instance of Each Resource Type

- If all the resources have only a single instance, then deadlock detection-algorithm can be defined using a wait-for-graph.
- The wait-for-graph is applicable to only a single instance of a resource type.
- A wait-for-graph (WAG) is a variation of the resource-allocation-graph.
- The wait-for-graph can be obtained from the resource-allocation-graph by
  → removing the resource nodes and
  → collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists if and only if the corresponding graph contains two edges
  1) $P_i \rightarrow R_q$ and
  2) $R_q \rightarrow P_j$.
- For example:
  Consider resource-allocation-graph shown in Figure 3.6
  Corresponding wait-for-graph is shown in Figure 3.7.

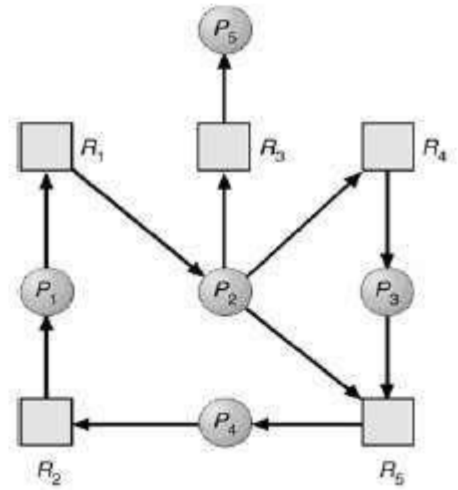


Figure 3.6 Resource-allocation-graph

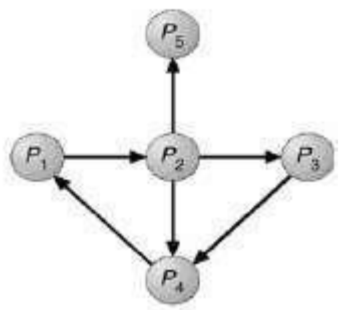


Figure 3.7 Corresponding wait-for-graph.

- A deadlock exists in the system if and only if the wait-for-graph contains a cycle.
- To detect deadlocks, the system needs to
  → maintain the wait-for-graph and
  → periodically execute an algorithm that searches for a cycle in the graph.

3.7.2 Several Instances of a Resource Type

- The wait-for-graph is applicable to only a single instance of a resource type.
- Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
- Solution: The following detection-algorithm can be used for a multiple instance of a resource type.
- Assumptions:

  Let 'n' be the number of processes in the system
  Let 'm' be the number of resources types.
- Following data structures are used to implement this algorithm.

  1) Available [m]
  - This vector indicates the no. of available resources of each type.
  - If Available[j]=k, then k instances of resource type Rj is available.

  2) Allocation [n][m]
  - This matrix indicates no. of resources currently allocated to each process.
  - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.

  3) Request [n][m]
  - This matrix indicates the current request of each process.
  - If Request [i, j] = k, then process Pi is requesting k more instances of resource type Rj.

```
Step 1:
        Let Work and Finish be vectors of length m and n respectively.
                a) Initialize Work = Available
                b)      For i=0,1,2…..... n
                        if Allocation(i) != 0
                        then
                                Finish[i] = false;
                        else
                                Finish[i] = true;
Step 2:
        Find an index(i) such that both
                a) Finish[i] = false
                b) Request(i) <= Work.
                If no such i exist, goto step 4.
Step 3:
                Set:
                    Work = Work + Allocation(i)
                    Finish[i] = true
        Go to step 2.
Step 4:
        If Finish[i] = false for some i where 0 < i < n, then the system is in a deadlock state.
```

3.7.3 Detection-Algorithm Usage

- The detection-algorithm must be executed based on following factors:

  1) The frequency of occurrence of a deadlock. 2) The no. of processes affected by the deadlock.
- If deadlocks occur frequently, then the detection-algorithm should be executed frequently.
- Resources allocated to deadlocked-processes will be idle until the deadlock is broken.
- Problem:

  Deadlock occurs only when some processes make a request that cannot be granted immediately.
- Solution 1:
  - The deadlock-algorithm must be executed whenever a request for allocation cannot be granted immediately.
  - In this case, we can identify

    → set of deadlocked-processes and
    → specific process causing the deadlock.
- Solution 2:
  - The deadlock-algorithm must be executed in periodic intervals.
  - For example:

    → once in an hour
    → whenever CPU utilization drops below certain threshold

3.8 Recovery from deadlock

- Three approaches to recovery from deadlock:
  1) Inform the system-operator for manual intervention.
  2) Terminate one or more deadlocked-processes.
  3) Preempt(or Block) some resources.

3.8.1 Process Termination

- Two methods to remove deadlocks:

  1) Terminate all deadlocked-processes.
  - ➢ This method will definitely break the deadlock-cycle.
  - ➢ However, this method incurs great expense. This is because
    - → Deadlocked-processes might have computed for a long time.
    - → Results of these partial computations must be discarded.
    - → Probably, the results must be re-computed later.

  2) Terminate one process at a time until the deadlock-cycle is eliminated.
  - ➢ This method incurs large overhead. This is because
    after each process is aborted,
    deadlock-algorithm must be executed to determine if any other process is still deadlocked
- For process termination, following factors need to be considered:
  1) The priority of process.
  2) The time taken by the process for computation & the required time for complete execution.
  3) The no. of resources used by the process.
  4) The no. of extra resources required by the process for complete execution.
  5) The no. of processes that need to be terminated for deadlock-free execution.
  6) The process is interactive or batch.

3.8.2 Resource Preemption

- Some resources are taken from one or more deadlocked-processes.
- These resources are given to other processes until the deadlock-cycle is broken.
- Three issues need to be considered:

  1) Selecting a victim
  - ➢ Which resources/processes are to be pre-empted (or blocked)?
  - ➢ The order of pre-emption must be determined to minimize cost.
  - ➢ Cost factors includes
    1. The time taken by deadlocked-process for computation.
    2. The no. of resources used by deadlocked-process.

  2) Rollback
  - ➢ If a resource is taken from a process, the process cannot continue its normal execution.
  - ➢ In this case, the process must be rolled-back to break the deadlock.
  - ➢ This method requires the system to keep more info. about the state of all running processes.

  3) Starvation
  - ➢ Problem: In a system where victim-selection is based on cost-factors, the same process may be always picked as a victim.
  - ➢ As a result, this process never completes its designated task.
  - ➢ Solution: Ensure a process is picked as a victim only a (small) finite number of times.

Exercise Problems

1) Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	0	2	0	0	4	1	0	2
P1	1	0	0	2	0	1			
P2	1	3	5	1	3	7			
P3	6	3	2	8	4	2			
P4	1	4	3	1	5	7			

Answer the following questions using Banker's algorithm:

i) What is the content of the matrix need?
ii) Is the system in a safe state?
iii) If a request from process P2 arrives for (0 0 2) can the request be granted immediately?

Solution (i):

• The content of the matrix Need is given by
Need = Max - Allocation

• So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P0	0	0	2
P1	1	0	1
P2	0	0	2
P3	2	1	0
P4	0	1	4

Solution (ii):

• Applying the Safety algorithm on the given system,

Step 1: Initialization
Work = Available i.e. Work =1 0 2
……P0………P1……..P2……..P3……P4…..
Finish = | false | false | false | false | false |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (0 0 2)<=(1 0 2) → true
So P0 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P0] =(1 0 2)+(0 0 2)=(1 0 4)
……P0………P1…….P2… ..... P3……P4….
Finish = | true | false | false | false | false |

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (1 0 1)<=(1 0 4) → true
So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(1 0 4)+(1 0 0)=(2 0 4)
…….P0…….P1……P2………P3… .... P4…
Finish = | true | true | false | false | false |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (0 0 2)<=(2 0 4) → true
So P2 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P2] =(2 0 4)+(1 3 5)=(3 3 9)
……P0…….P1…….P2…..... P3……P4….
Finish = | true | true | true | false | false |

Step 2: For i=3
Finish[P3] = false and Need[P3]<=Work i.e. (2 1 0)<=(3 3 9) ➔ true
So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (3 3 9)+(6 3 2)=(9 6 11)
…..P0……..P1……P2……..P3…….P4….
Finish = | true | true | true | true | false |

Step 2: For i=4
Finish[P4] = false and Need[P4]<=Work i.e. (0 1 4)<=(9 6 11) ➔ true
So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(9 6 11)+(1 4 3)=(10 10 14)
……P0…….P1……..P2……P3…….P4….
Finish = | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is currently in a safe state.
The safe sequence is <P0, P1, P2, P3, P4>.

Conclusion: Yes, the system is currently in a safe state.

Solution (iii): P2 requests (0 0 2) i.e. Request[P2]=0 0 2

- To decide whether the request is granted, we use Resource Request algorithm.

Step 1: Request[P2]<=Need[P2] i.e. (0 0 2) <= (1 3 7) ➔ true.
Step 2: Request[P2]<=Available i.e. (0 0 2) <= (1 0 2) ➔ true.
Step 3: Available = Available – Request[P2] = (1 0 2) - (0 0 2)= (1 0 0)
Allocation[P2] = Allocation[P2] + Request[P2] = (1 3 5) + (0 0 2)= (1 3 7)
Need[P2] = Need[P2] – Request[P2] = (0 0 2) - (0 0 2)= (0 0 0)

- We arrive at the following new system state:

	Allocation			**Max**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	0	2	0	0	4	1	0	0
P1	1	0	0	2	0	1			
P2	1	3	7	1	3	7			
P3	6	3	2	8	4	2			
P4	1	4	3	1	5	7			

- The content of the matrix Need is given by
  Need = Max - Allocation
- So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P0	0	0	2
P1	1	0	1
P2	0	0	0
P3	2	1	0
P4	0	1	4

- To determine whether this new system state is safe, we again execute Safety algorithm.

Step 1: Initialization
Work = Available i.e. Work =2 3 0
....P0………P1…….P2……….P3…..P4…..
Finish = | false | false | false | false | false |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (0 0 2)<=(2 3 0) ➔ false
So P0 must wait.

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (1 0 1)<=(2 3 0) → false
So P1 must wait.

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (0 0 0)<=(2 3 0) → true
So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(1 0 0)+(1 3 7)=(2 3 7)
.....P0…….P1……..P2…..P3....... P4….
Finish = | false | false | true | false | false |

Step 2: For i=3
Finish[P3] = false and Need[P3]<=Work i.e. (2 1 0)<=(2 3 7) → true
So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (2 3 7)+(6 3 2)=(8 6 9)
....P0……….P1…….P2…….P3…....P4…
Finish = | false | false | true | true | false |

Step 2: For i=4
Finish[P4] = false and Need[P4]<=Work i.e. (0 1 4)<=(8 6 9) → true
So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(8 6 9)+(0 1 4)=(8 7 13)
....P0……….P1…….P2…….P3…....P4…
Finish = | false | false | true | true | true |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (0 0 2)<=(8 7 13) → true
So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P0] =(8 7 13)+(0 0 2)=(8 7 15)
....P0……….P1…….P2…….P3…….P4…
Finish = | true | false | true | true | true |

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (1 0 1)<=(8 7 15) → true
So P1 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P1] =(8 7 15)+(1 0 0)=(9 7 15)
....P0……….P1…….P2……P3…….P4…
Finish = | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is in a safe state.
The safe sequence is <P2, P3, P4, P0, P1>.

Conclusion: Since the system is in safe sate, the request can be granted.

2) For the following snapshot, find the safe sequence using Banker's algorithm:
The number of resource units is (A, B, C) which are (7, 7, 10) respectively.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	2	2	3	3	6	8	7	7	10
P2	2	0	3	4	3	3			
P3	1	2	4	3	4	4			

Solution:

• The content of the matrix Need is given by
Need = Max - Allocation

• So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P1	1	4	5
P2	2	3	0
P3	2	2	0

•

Applying the Safety algorithm on the given system, Step 1: Initialization

Here, m=3, n=3
Work = Available i.e. Work =7 7 10
….P1………..P2…....P3…
Finish = | false | false | false |

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (1 4 5)<=(7 7 10) ➔ true
So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(7 7 10)+(2 2 3)=(9 9 13)
……P1……P2……….P3….
Finish = | true | false | false |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (2 3 0) <=(9 9 13) ➔ true
So P2 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P2] =(9 9 13)+(2 0 3)=(11 9 16)
…..P1……P2…….P3……
Finish = | true | true | false |

Step 2: For i=3
Finish[P3] = false and Need[P3]<=Work i.e. (2 2 0)<=(11 9 16) ➔ true
So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (11 9 16)+(1 2 4)=(12 11 20)
……P1……P2…….P3….
Finish = | true | true | true |

Step 4: Finish[Pi] = true for 1<=i<=3
Hence, the system is currently in a safe state.
The safe sequence is <P1, P2, P3>.

Conclusion: Yes, the system is currently in a safe state.

3) Consider the following snapshot of resource-allocation at time t1.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

i) What is the content of the matrix need?
ii) Show that the system is not deadlock by generating one safe sequence
iii) At instance t, P2 makes one additional for instance of type C. Show that the system is deadlocked if the request is granted. Write down deadlocked-processes.

Solution (i):

• The content of the matrix Need is given by
Need = Max - Allocation
• So, the content of Need Matrix is:

	Need		
	A	**B**	**C**
P0	0	0	0
P1	0	0	2
P2	0	0	0
P3	0	0	0
P4	0	0	0

Solution (ii):

• Applying the Safety algorithm on the given system,

Step 1: Initialization
Work = Available i.e. Work =0 0 0
……P0………P1…….P2…..... P3……P4…
Finish = | false | false | false | false | false |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (0 0 0)<=(0 0 0) ➔ true
So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P0] =(0 0 0)+(0 1 0)=(0 1 0)
…..P0………P1…….P2…….P3… ... P4…
Finish = | true | false | false | false | false |

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (0 0 2)<=(0 1 0) ➔ false
So P1 must wait.

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (0 0 0) <=(0 1 0) ➔ true
So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(0 1 0)+(3 0 3)=(5 1 3)
....P0……P1…….P2……….P3…….P4…
Finish = | true | false | true | false | false |

Step 2: For i=3
Finish[P3] = false and Need[P3]<=Work i.e. (0 0 0)<=(5 1 3) ➔ true
So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (5 1 3)+(2 1 1)=(5 2 4)
…P0……P1………P2…….P3…….P4…
Finish = | true | false | true | true | false |

Step 2: For i=4
Finish[P4] = false and Need[P4]<=Work i.e. (0 0 0)<=(5 2 4) ➔ true
So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(5 2 4)+(0 0 2)=(5 2 6)
....P0……P1…….P2…….P3…….P4….
Finish = | true | false | true | true | true |

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (0 0 2)<=(5 2 6) ➔ true
So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P1] =(5 2 6)+(2 0 0)=(7 2 6)
...P0…….P1……..P2…….P3…….P4…
Finish= | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is currently in a safe state.
The safe sequence is <P0, P2, P3, P4, P1>.

Conclusion: Yes, the system is currently in a safe state. Hence there is no deadlock in the system.

Solution (iii): P2 requests (0 0 1) i.e. Request[P1]=0 0 1
• To decide whether the request is granted, we use Resource Request algorithm.
Step 1: Request[P1]<=Need[P1] i.e. (0 0 1)<=(0 0 2) ➔ true.
Step 2: Request[P1]<=Available i.e. (0 0 1)<=(0 0 0) ➔ false.

Conclusion: Since Request[P1]>Available, we cannot process this request.
Since P2 will be in waiting state, deadlock occurs in the system.

4) For the given snapshot :

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	1	2	0	0	1	2	1	5	2	0
P2	1	0	0	0	1	7	5	0				
P3	1	3	5	4	2	3	5	6				
P4	0	6	3	2	0	6	5	2				
P5	0	0	1	4	0	6	5	6				

Using Banker's algorithm:

i) What is the need matrix content?

ii) Is the system in safe state?

iii) If a request from process P2(0,4,2,0) arrives, can it be granted?

Solution (i):

• The content of the matrix Need is given by

Need = Max - Allocation

• So, the content of Need Matrix is:

	Need			
	A	**B**	**C**	**D**
P1	0	0	0	0
P2	0	7	5	2
P3	1	0	0	2
P4	0	0	2	0
P5	0	6	4	2

Solution (ii):

• Applying the Safety algorithm on the given system,

Step 1: Initialization

Work = Available i.e. Work =1 5 2 0

....P1………P2…….P3……….P4…..P5…..

Finish = | false | false | false | false | false |

Step 2: For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (0 0 0 0)<=(1 5 2 0) ➔ true

So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(1 5 2 0)+(0 0 1 2)=(1 5 3 2)

....P1………P2…….P3…….P4…....P5…

Finish = | true | false | false | false | false |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (0 7 5 2)<=(1 5 3 2) ➔ false

So P2 must wait.

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (1 0 0 2)<=(1 5 3 2) ➔ true

So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (1 5 3 2)+(1 3 5 4)=(2 8 8 6)

....P1………P2…….P3…….P4…....P5…

Finish = | true | false | true | false | false |

Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (0 0 2 0)<=(2 8 8 6) ➔ true

So P4 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(2 8 8 6)+(0 6 3 2)=(2 14 11 8)

....P1………P2…….P3…….P4…….P5…

Finish = | true | false | true | true | false |

Step 2: For i=5
Finish[P5] = false and Need[P5]<=Work i.e. (0 6 4 2)<=(2 14 11 8) ➔ true
So P5 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P5] =(2 14 11 8)+(0 0 1 4)=(2 14 12 12)
....P1………P2…….P3…….P4……P5…
Finish = | true | false | true | true | true |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (0 7 5 2) <=(2 14 12 12) ➔ true
So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(2 14 12 12)+(1 0 0 0)=(3 14 12 12)
....P1………P2…….P3…….P4……P5…
Finish = | true | true | true | true | true |

Step 4: Finish[Pi] = true for 1<=i<=5
Hence, the system is currently in a safe state.
The safe sequence is <P1, P3, P4, P5, P2>.

Conclusion: Yes, the system is currently in a safe state.

Solution (iii): P2 requests (0 4 2 0) i.e. Request[P2]= 0 4 2 0

• To decide whether the request is granted, we use Resource Request algorithm.
Step 1: Request[P2]<=Need[P2] i.e. (0 4 2 0) <= (0 7 5 2) ➔ true.
Step 2: Request[P2]<=Available i.e. (0 4 2 0) <= (1 5 2 0) ➔ true.
Step 3: Available = Available – Request[P2] = (1 5 2 0) - (0 4 2 0)= (1 1 0 0)
Allocation[P2] = Allocation[P2] + Request[P2] = (1 0 0 0) + (0 4 2 0)= (1 4 2 0)
Need[P2] = Need[P2] – Request[P2] = (0 7 5 2) - (0 4 2 0)= (0 3 3 2)

• We arrive at the following new system state

	Allocation				**Max**				**Available**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P1	0	0	1	2	0	0	1	2	1	1	0	0
P2	1	4	2	0	1	7	5	0				
P3	1	3	5	4	2	3	5	6				
P4	0	6	3	2	0	6	5	2				
P5	0	0	1	4	0	6	5	6				

• The content of the matrix Need is given by
Need = Max - Allocation

• So, the content of Need Matrix is:

	Need			
	A	**B**	**C**	**D**
P1	0	0	0	0
P2	0	3	3	2
P3	1	0	0	2
P4	0	0	2	0
P5	0	6	4	2

• Applying the Safety algorithm on the given system,
Step 1: Initialization
Work = Available i.e. Work =1 1 0 0
....P1………P2…….P3……….P4… ..P5….
Finish = | false | false | false | false | false |

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (0 0 0 0)<=(1 1 0 0) ➔ true
So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(1 1 0 0)+(0 0 1 2)=(1 1 1 2)

....P1………P2…….P3……..P4......P5…

Finish = | true | false | false | false | false |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (0 3 3 2)<=(1 1 1 2) ➔false

So P2 must wait.

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (1 0 0 2)<=(1 1 1 2) ➔ true

So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (1 1 1 2)+(1 3 5 4)=(2 4 6 6)

....P1……P2……….P3…….P4…....P5….

Finish = | true | false | true | false | false |

Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (0 0 2 0)<=(2 4 6 6) ➔ true

So P4 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(2 4 6 6)+(0 6 3 2)=(2 10 9 8)

....P1………P2…….P3…….P4…….P5….

Finish = | true | false | true | true | false |

Step 2: For i=5

Finish[P5] = false and Need[P5]<=Work i.e. (0 6 4 2)<=(2 10 9 8) ➔ true

So P5 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P5] =(2 10 9 8)+(0 0 1 4)=(2 10 10 12)

....P1………P2…….P3…….P4……P5….

Finish = | true | false | true | true | true |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (0 3 3 2) <=(2 10 10 12) ➔ true

So P2 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P2] =(2 10 10 12)+(1 4 2 0)=(3 14 12 12)

....P1………P2……P3…….P4……P5….

Finish = | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4

Hence, the system is currently in a safe state.

The safe sequence is <P1, P3, P4, P5, P2>.

Conclusion: Since the system is in safe sate, the request can be granted.

5) Consider a system containing 'm' resources of the same type being shared by 'n' processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:

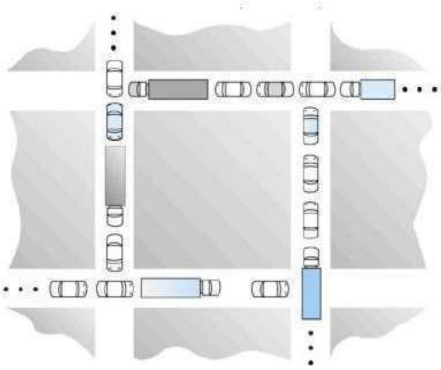
i) The maximum need of each process is between 1 and m resources

ii) The sum of all maximum needs is less than m+n.

Ans:

- Suppose N = Sum of all $Need_i$
  A = Sum of all $Allocation_i$
  M = Sum of all Max_i.
- Use contradiction to prove: Assume this system is not deadlock free.
- If there exists a deadlock state, then A=m because there's only one kind of resource and resources can be requested and released only one at a time.
- From condition (ii), N+A = M<m+n
- So we get N+m <m +n.
- So we get N < n.
- It shows that at least one process i that $Need_i$=0.
- From condition (i), P_i can release at least one resource.
- So, there are n-1 processes sharing 'm' resources now, condition (i) and (ii) still hold.
- Go on the argument, no process will wait permanently, so there's no deadlock.

6) Consider the traffic deadlock depicted in the figure given below, explain that the four necessary conditions for dead lock indeed hold in this examples.



Ans:

- The four necessary conditions for a deadlock are:
  1) Mutual exclusion
  2) Hold-and-wait
  3) No preemption and
  4) Circular-wait.
- The mutual exclusion condition holds since only one car can occupy a space in the roadway.
- Hold-and-wait occurs where a car holds onto its place in the roadway while it waits to advance in the roadway.
- A car cannot be removed (i.e. preempted) from its position in the roadway.
- Lastly, there is indeed a circular-wait as each car is waiting for a subsequent car to advance.
- The circular-wait condition is also easily observed from the graphic.

MODULE 3 (CONT.): MEMORY MANAGEMENT

3.9 Main Memory
3.9.1 Basic Hardware
• Program must be
→ brought (from disk) into memory and
→ placed within a process for it to be run.
• Main-memory and registers are only storage CPU can access directly.
• Register access in one CPU clock.
• Main-memory can take many cycles.
• Cache sits between main-memory and CPU registers.
• Protection of memory required to ensure correct operation.
• A pair of base- and limit-registers define the logical (virtual) address space (Figure 3.8 & 3.9).

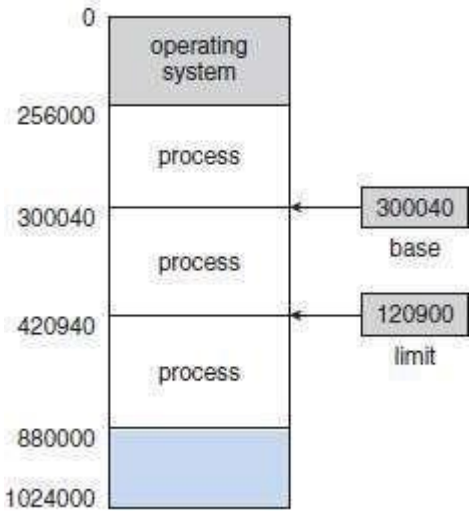


Figure 3.8 A base and a limit-register define a logical-address space

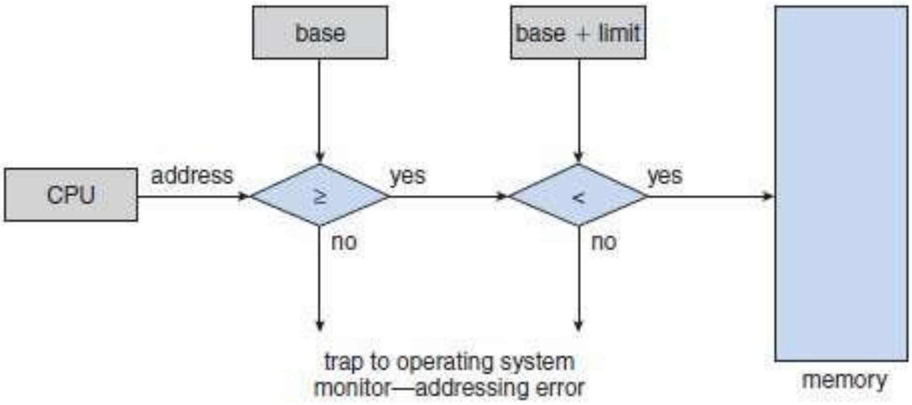


Figure 3.9 Hardware address protection with base and limit-registers

3.9.2 Address Binding

• Address binding of instructions to memory-addresses can happen at 3 different stages (Figure 3.10):

1) Compile Time

➢ If memory-location known a priori, absolute code can be generated.

➢ Must recompile code if starting location changes.

2) Load Time

➢ Must generate relocatable code if memory-location is not known at compile time.

3) Execution Time

➢ Binding delayed until run-time if the process can be moved during its execution from one memory-segment to another.

➢ Need hardware support for address maps (e.g. base and limit-registers).

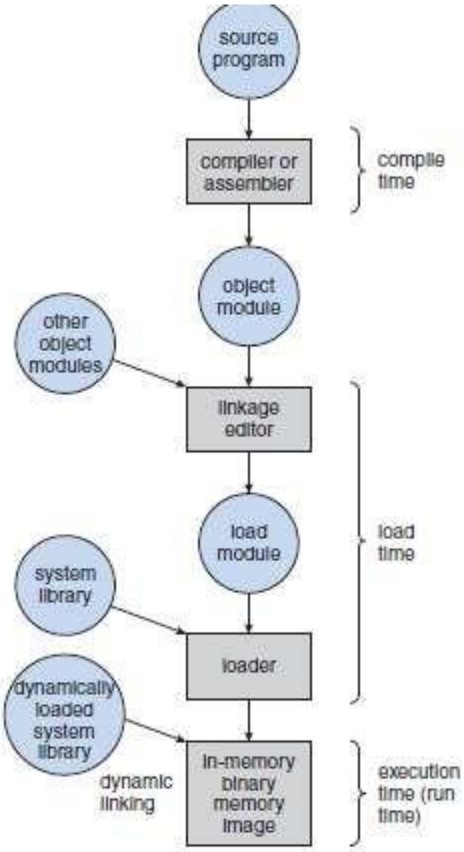


Figure 3.10 Multistep processing of a user-program

3.9.3 Logical versus Physical Address Space

• **Logical-address** is generated by the CPU (also referred to as virtual-address).
  Physical-address is the address seen by the memory-unit.

• Logical & physical-addresses are the same in compile-time & load-time address-binding methods.
  Logical and physical-addresses differ in execution-time address-binding method.

• MMU (Memory-Management Unit)
  - ➢ Hardware device that maps virtual-address to physical-address (Figure 3.11).
  - ➢ The value in the relocation-register is added to every address generated by a user-process at the time it is sent to memory.
  - ➢ The user-program deals with logical-addresses; it never sees the real physical-addresses.

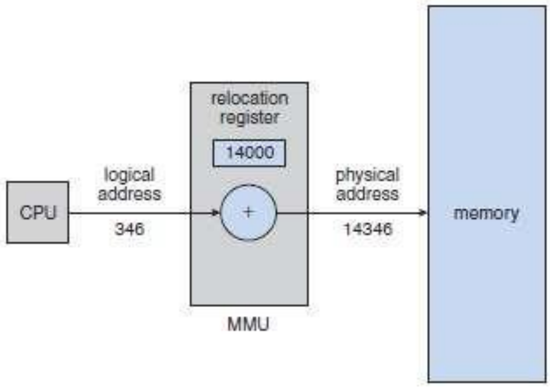


Figure 3.11 Dynamic relocation using a relocation-register

3.9.4 Dynamic Loading

• This can be used to obtain better memory-space utilization.
• A routine is not loaded until it is called.
• This works as follows:
  1) Initially, all routines are kept on disk in a relocatable-load format.
  2) Firstly, the main-program is loaded into memory and is executed.
  3) When a main-program calls the routine, the main-program first checks to see whether the routine has been loaded.
  4) If routine has been not yet loaded, the loader is called to load desired routine into memory.
  5) Finally, control is passed to the newly loaded-routine.
• Advantages:
  1) An unused routine is never loaded.
  2) Useful when large amounts of code are needed to handle infrequently occurring cases.
  3) Although the total program-size may be large, the portion that is used (and hence loaded) may be much smaller.
  4) Does not require special support from the OS.

3.9.5 Dynamic Linking and Shared Libraries

• Linking postponed until execution-time.
• This feature is usually used with system libraries, such as language subroutine libraries.
• A stub is included in the image for each library-routine reference.
• The **stub** is a small piece of code used to locate the appropriate memory-resident library-routine.
• When the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
• Stub replaces itself with the address of the routine, and executes the routine.
• Thus, the next time that particular code-segment is reached, the library-routine is executed directly, incurring no cost for dynamic-linking.
• All processes that use a language library execute only one copy of the library code.

Shared libraries

• A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.
• Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

3.10 Swapping

• A process must be in memory to be executed.
• A process can be
→ swapped temporarily out-of-memory to a backing-store and
→ then brought into memory for continued execution.
• **Backing-store** is a fast disk which is large enough to accommodate copies of all memory-images for all users.
• **Roll out/Roll in** is a swapping variant used for priority-based scheduling algorithms.
➢ Lower-priority process is swapped out so that higher-priority process can be loaded and executed.
➢ Once the higher-priority process finishes, the lower-priority process can be swapped back in and continued (Figure 3.12).

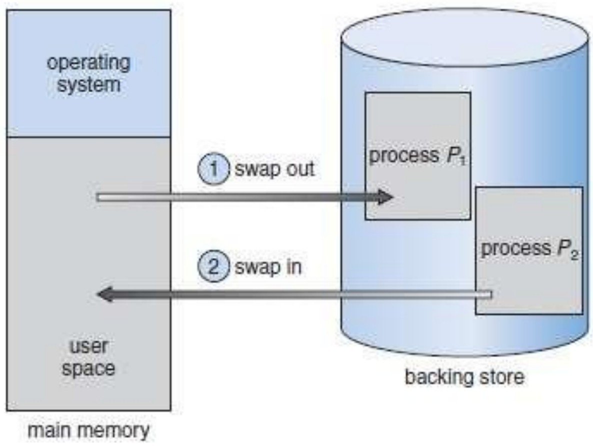


Figure 3.12 Swapping of two processes using a disk as a backing-store

• Swapping depends upon address-binding:
1) If binding is done at load-time, then process cannot be easily moved to a different location.
2) If binding is done at execution-time, then a process can be swapped into a different memory-space, because the physical-addresses are computed during execution-time.
• Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to the amount of memory swapped.
• Disadvantages:
1) Context-switch time is fairly high.
2) If we want to swap a process, we must be sure that it is completely idle.
Two solutions:
i) Never swap a process with pending I/O.
ii) Execute I/O operations only into OS buffers.

3.11 Contiguous Memory Allocation

• Memory is usually divided into 2 partitions:
  → One for the resident OS.
  → One for the user-processes.
• Each process is contained in a single contiguous section of memory.

3.11.1 Memory Mapping & Protection

• Memory-protection means
  → protecting OS from user-process and
  → protecting user-processes from one another.
• Memory-protection is done using
  → **Relocation-register**: contains the value of the smallest physical-address.
  → **Limit-register**: contains the range of logical-addresses.
• Each logical-address must be less than the limit-register.
• The MMU maps the logical-address dynamically by adding the value in the relocation-register. This mapped-address is sent to memory (Figure 3.13).
• When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.
• Because every address generated by the CPU is checked against these registers, we can protect the OS from the running-process.
• The relocation-register scheme provides an effective way to allow the OS size to change dynamically.
• **Transient OS code**: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

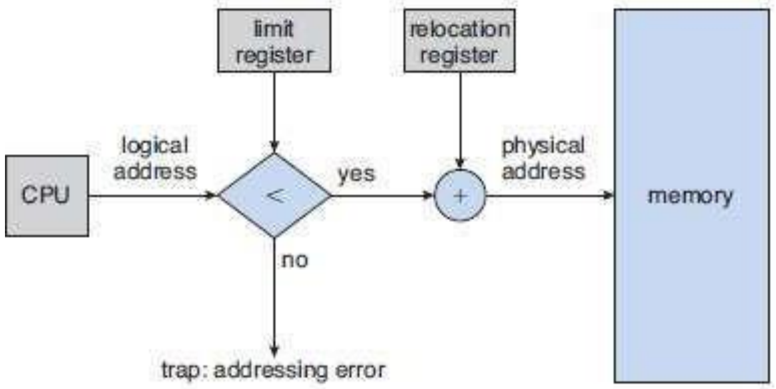


Figure 3.13 Hardware support for relocation and limit-registers

3.11.2 Memory Allocation

• Two types of memory partitioning are: 1) Fixed-sized partitioning and
2) Variable-sized partitioning

1) Fixed-sized Partitioning

➢ The memory is divided into fixed-sized partitions.
➢ Each partition may contain exactly one process.
➢ The degree of multiprogramming is bound by the number of partitions.
➢ When a partition is free, a process is
→ selected from the input queue and
→ loaded into the free partition.
➢ When the process terminates, the partition becomes available for another process.

2) Variable-sized Partitioning

➢ The OS keeps a table indicating
→ which parts of memory are available and
→ which parts are occupied.
➢ A **hole** is a block of available memory.
➢ Normally, memory contains a set of holes of various sizes.
➢ Initially, all memory is
→ available for user-processes and
→ considered one large hole.
➢ When a process arrives, the process is allocated memory from a large hole.
➢ If we find the hole, we
→ allocate only as much memory as is needed and
→ keep the remaining memory available to satisfy future requests.

• Three strategies used to select a free hole from the set of available holes.

1) First Fit

➢ Allocate the first hole that is big enough.
➢ Searching can start either
→ at the beginning of the set of holes or
→ at the location where the previous first-fit search ended.

2) Best Fit

➢ Allocate the smallest hole that is big enough.
➢ We must search the entire list, unless the list is ordered by size.
➢ This strategy produces the smallest leftover hole.

3) Worst Fit

➢ Allocate the largest hole.
➢ Again, we must search the entire list, unless it is sorted by size.
➢ This strategy produces the largest leftover hole.

• First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

3.11.3 Fragmentation

• Two types of memory fragmentation: 1) Internal fragmentation and
2) External fragmentation

1) Internal Fragmentation

• The general approach is to
→ break the physical-memory into fixed-sized blocks and
→ allocate memory in units based on block size (Figure 3.14).

• The allocated-memory to a process may be slightly larger than the requested-memory.

• The difference between requested-memory and allocated-memory is called internal fragmentation i.e. Unused memory that is internal to a partition.

2) External Fragmentation

• External fragmentation occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).

• Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.

• Statistical analysis of first-fit reveals that
→ given N allocated blocks, another 0.5 N blocks will be lost to fragmentation.
This property is known as the **50-percent rule**.

• Two solutions to external fragmentation (Figure 3.15):

1) Compaction

➢ The goal is to shuffle the memory-contents to place all free memory together in one large hole.

➢ Compaction is possible only if relocation is
→ dynamic and
→ done at execution-time.

2) Permit the logical-address space of the processes to be non-contiguous.

➢ This allows a process to be allocated physical-memory wherever such memory is available.

➢ Two techniques achieve this solution:
1) Paging and
2) Segmentation.

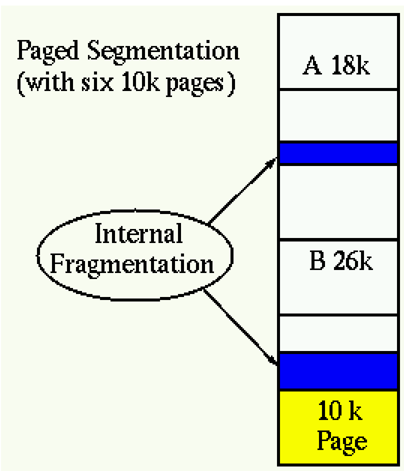


Figure 3.14: Internal fragmentation

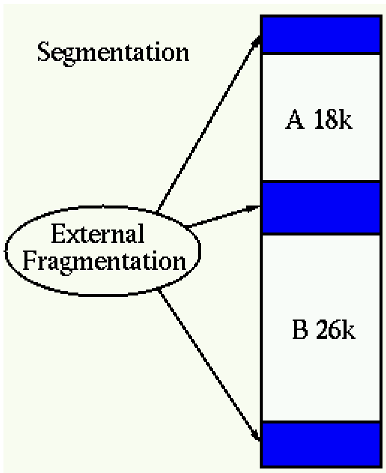


Figure 3.15: External fragmentation

3.13 Paging

- Paging is a memory-management scheme.
- This permits the physical-address space of a process to be non-contiguous.
- This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.
- Traditionally: Support for paging has been handled by hardware.
  Recent designs: The hardware & OS are closely integrated.

3.13.1 Basic Method

- Physical-memory is broken into fixed-sized blocks called **frames**(Figure 3.16).
  Logical-memory is broken into same-sized blocks called **pages.**
- When a process is to be executed, its pages are loaded into any available memory-frames from the backing-store.
- The backing-store is divided into fixed-sized blocks that are of the same size as the memory-frames.

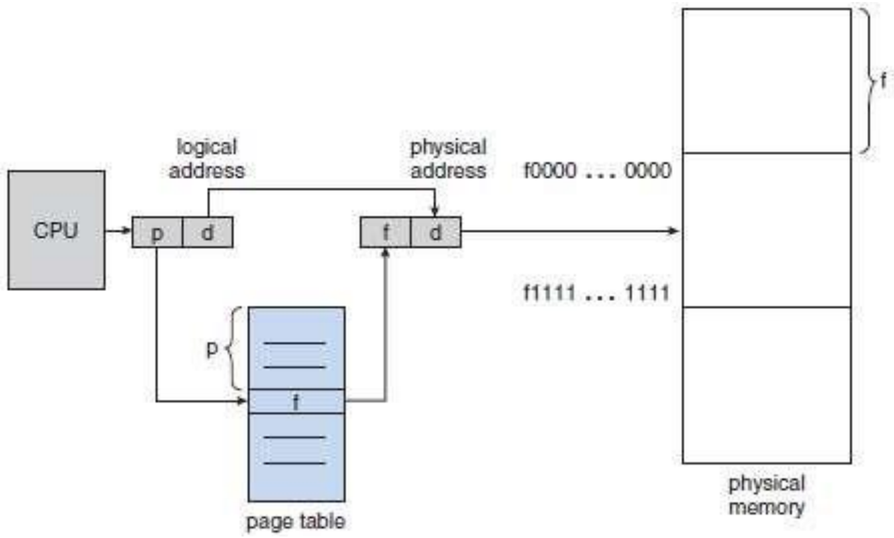


Figure 3.16 Paging hardware

- The page-table contains the base-address of each page in physical-memory.
- Address generated by CPU is divided into 2 parts (Figure 3.17):
  1) Page-number(p) is used as an index to the page-table and
  2) Offset(d) is combined with the base-address to define the physical-address.
  This physical-address is sent to the memory-unit.

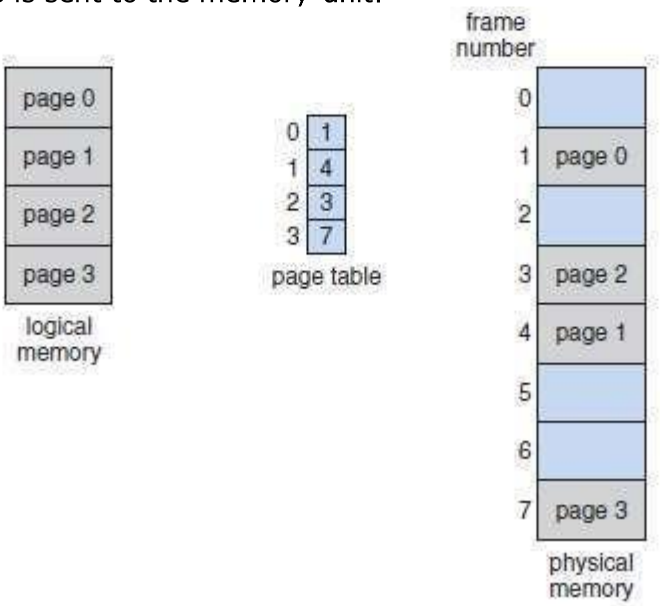
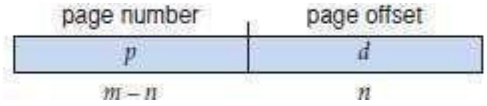


Figure 3.17 Paging model of logical and physical-memory

• The page-size (like the frame size) is defined by the hardware (Figure 3.18).
• If the size of the logical-address space is 2^m, and a page-size is 2^n addressing-units (bytes or words) then the high-order m-n bits of a logical-address designate the page-number, and the n low-order bits designate the page-offset.



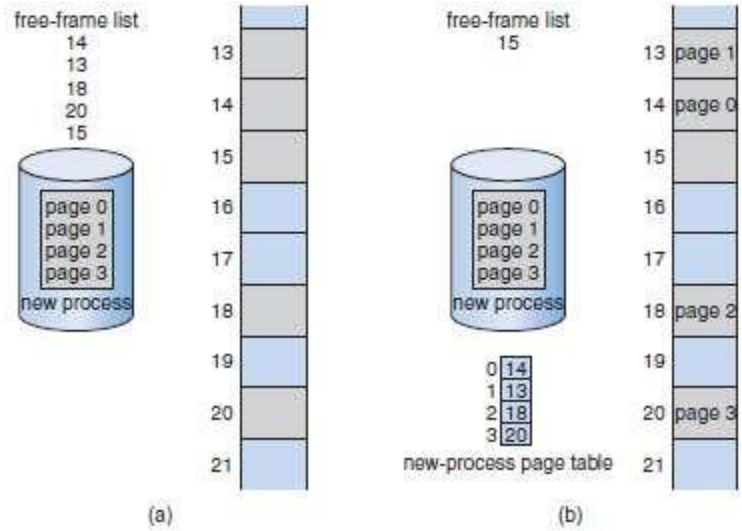


Figure 3.18 Free frames (a) before allocation and (b) after allocation

3.13.2 Hardware Support for Paging

- Most OS's store a page-table for each process.
- A pointer to the page-table is stored in the PCB.

Translation Lookaside Buffer

- The TLB is associative, high-speed memory.
- The TLB contains only a few of the page-table entries.
- Working:
  - ➢ When a logical-address is generated by the CPU, its page-number is presented to the TLB.
  - ➢ If the page-number is found (**TLB hit**), its frame-number is
    - → immediately available and
    - → used to access memory.
  - ➢ If page-number is not in TLB (**TLB miss**), a memory-reference to page table must be made.
  - ➢ The obtained frame-number can be used to access memory (Figure 3.19).
  - ➢ In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called **hit ratio**.
- Advantage: Search operation is fast.
  Disadvantage: Hardware is expensive.
- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely
  - → identify each process and
  - → provide address space protection for that process.

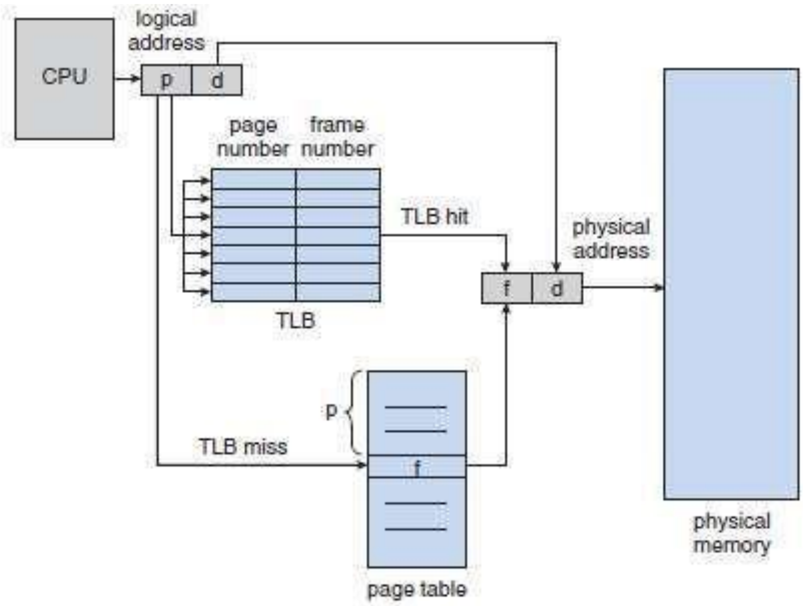


Figure 3.19 Paging hardware with TLB

3.13.3 Protection

• Memory-protection is achieved by **protection-bits** for each frame.
• The protection-bits are kept in the page-table.
• One protection-bit can define a page to be read-write or read-only.
• Every reference to memory goes through the page-table to find the correct frame-number.
• Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify that no writes are being made to a read-only page.
• An attempt to write to a read-only page causes a hardware-trap to the OS (or memory-protection violation).

Valid Invalid Bit

• This bit is attached to each entry in the page-table (Figure 3.20).

   1) Valid bit: The page is in the process' logical-address space.

   2) Invalid bit: The page is not in the process' logical-address space.

• Illegal addresses are trapped by use of valid-invalid bit.
• The OS sets this bit for each page to allow or disallow access to the page.

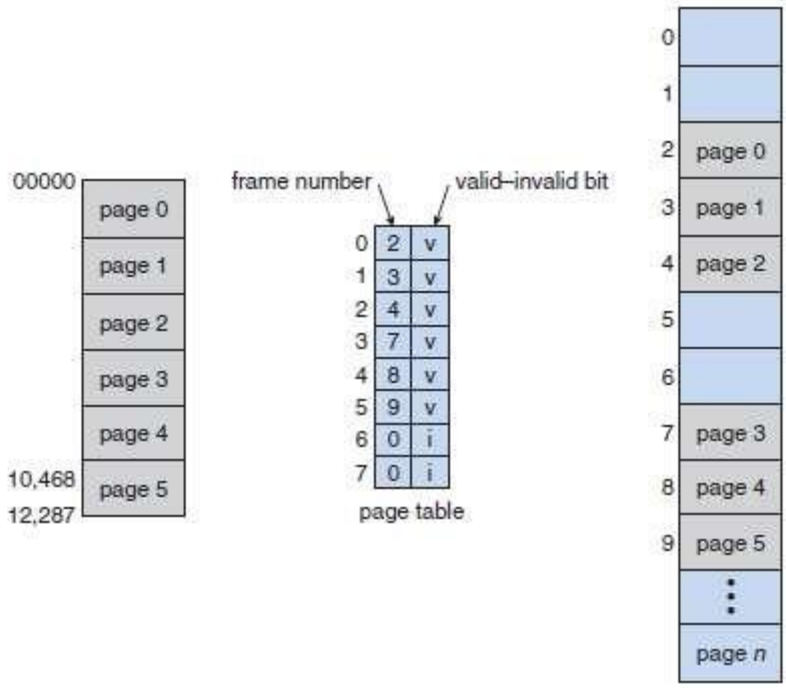


Figure 3.20 Valid (v) or invalid (i) bit in a page-table

3.13.4 Shared Pages

- Advantage of paging:
  1) Possible to share common code.
- Re-entrant code is non-self-modifying code, it never changes during execution.
- Two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data-storage to hold the data for the process's execution.
- The data for 2 different processes will be different.
- Only one copy of the editor need be kept in physical-memory (Figure 3.21).
- Each user's page-table maps onto the same physical copy of the editor,
  but data pages are mapped onto different frames.
- Disadvantage:
  1) Systems that use inverted page-tables have difficulty implementing shared-memory.

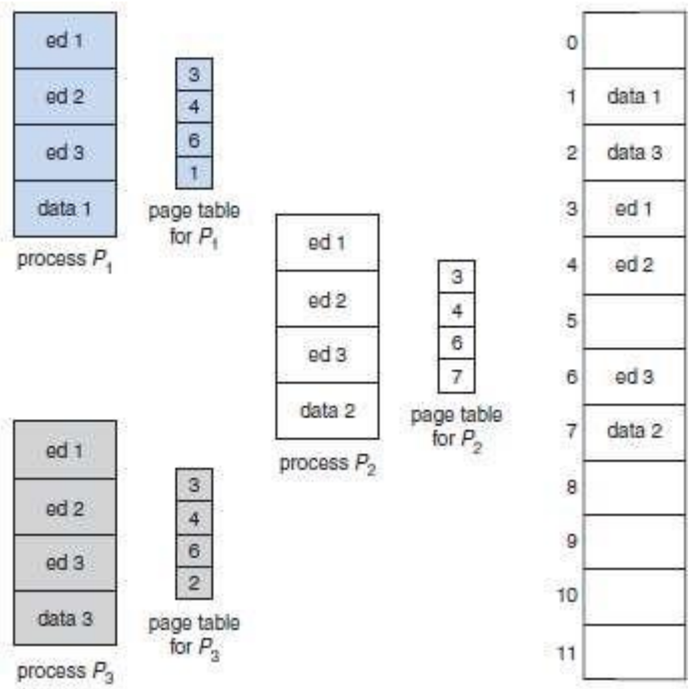


Figure 3.21 Sharing of code in a paging environment

3.14 Structure of the Page Table

1) Hierarchical Paging
2) Hashed Page-tables
3) Inverted Page-tables

3.14.1 Hierarchical Paging

• Problem: Most computers support a large logical-address space (2^{32} to 2^{64}). In these systems, the page-table itself becomes excessively large.
Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm

• The page-table itself is also paged (Figure 3.22).
• This is also known as a forward-mapped page-table because address translation works from the outer page-table inwards.
• For example (Figure 3.23):
  ➢ Consider the system with a 32-bit logical-address space and a page-size of 4 KB.
  ➢ A logical-address is divided into
    → 20-bit page-number and
    → 12-bit page-offset.
  ➢ Since the page-table is paged, the page-number is further divided into
    → 10-bit page-number and
    → 10-bit page-offset.
  ➢ Thus, a logical-address is as follows:

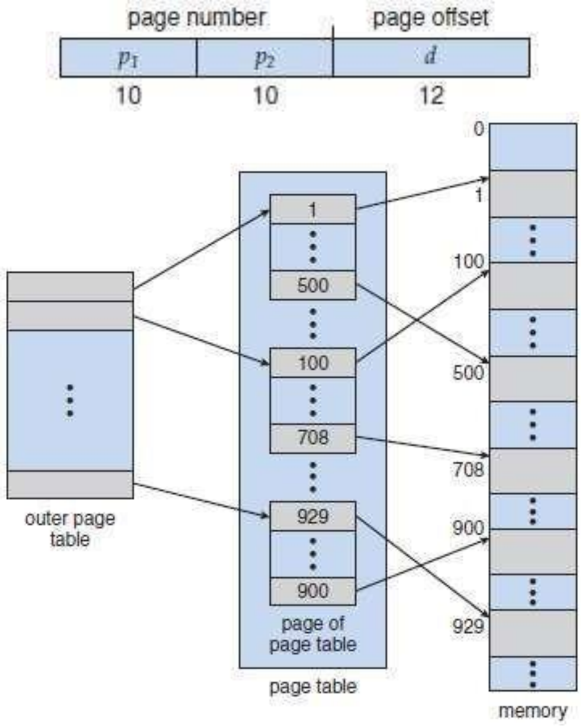


Figure 3.22 A two-level page-table scheme

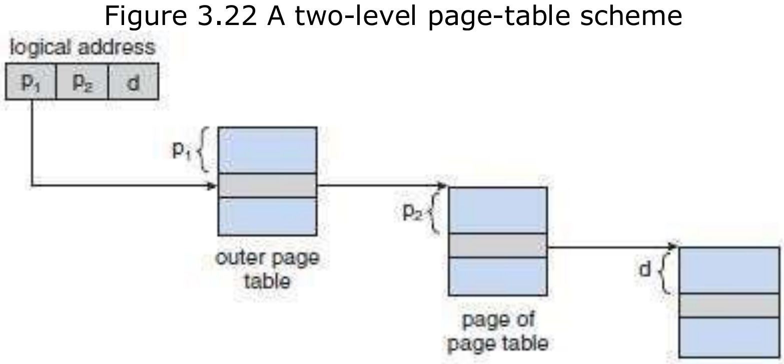


Figure 3.23 Address translation for a two-level 32-bit paging architecture

3.14.2 Hashed Page Tables

• This approach is used for handling address spaces larger than 32 bits.
• The hash-value is the virtual page-number.
• Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
• Each element consists of 3 fields:

 1) Virtual page-number
 2) Value of the mapped page-frame and
 3) Pointer to the next element in the linked-list.

• The algorithm works as follows (Figure 3.24):

 1) The virtual page-number is hashed into the hash-table.
 2) The virtual page-number is compared with the first element in the linked-list.
 3) If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
 4) If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.

Clustered Page Tables

• These are similar to hashed page-tables except that each entry in the hash-table refers to several pages rather than a single page.
• Advantages:

 1) Favorable for 64-bit address spaces.
 2) Useful for address spaces, where memory-references are noncontiguous and scattered throughout the address space.

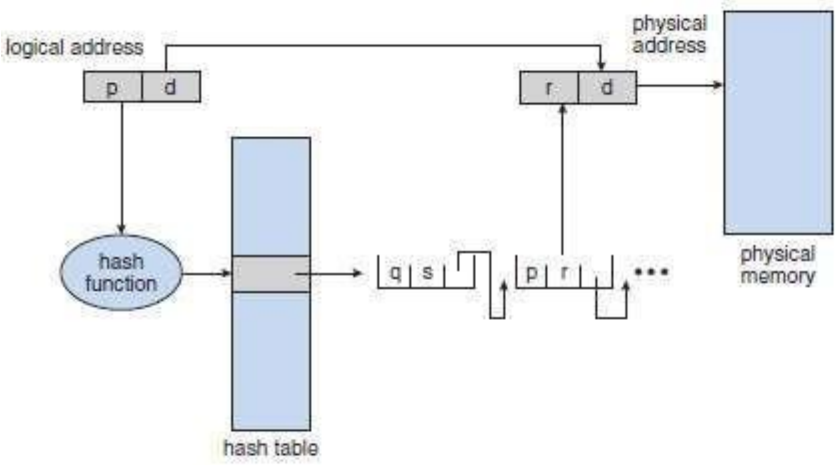


Figure 3.24 Hashed page-table

3.14.3 Inverted Page Tables

• Has one entry for each real page of memory.
• Each entry consists of
→ virtual-address of the page stored in that real memory-location and
→ information about the process that owns the page.

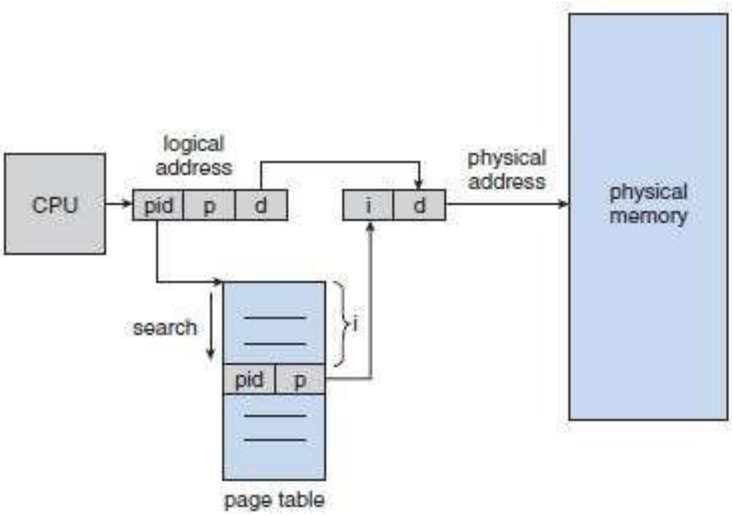


Figure 3.25 Inverted page-table

• Each virtual-address consists of a triplet (Figure 3.25):
<process-id, page-number, offset>.
• Each inverted page-table entry is a pair <process-id, page-number>
• The algorithm works as follows:
1) When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.
2) The inverted page-table is then searched for a match.
3) If a match is found, at entry i-then the physical-address <i, offset> is generated.
4) If no match is found, then an illegal address access has been attempted.
• Advantage:
1) Decreases memory needed to store each page-table
• Disadvantages:
1) Increases amount of time needed to search table when a page reference occurs.
2) Difficulty implementing shared-memory.

3.15 Segmentation

3.15.1 Basic Method

• This is a memory-management scheme that supports user-view of memory(Figure 3.26).
• A logical-address space is a collection of segments.
• Each segment has a name and a length.
• The addresses specify both
→ segment-name and
→ offset within the segment.
• Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.
For ex:
→ The code
→ Global variables
→ The heap, from which memory is allocated
→ The stacks used by each thread
→ The standard C library

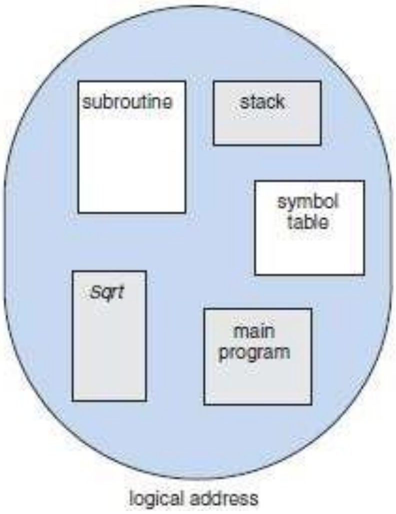


Figure 3.26 Programmer's view of a program

3.15.2 Hardware Support

• Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical-addresses.
• In the segment-table, each entry has following 2 fields:
1) Segment-base contains starting physical-address where the segment resides in memory.
2) Segment-limit specifies the length of the segment (Figure 3.27).
• A logical-address consists of 2 parts:
1) Segment-number(s) is used as an index to the segment-table .
2) Offset(d) must be between 0 and the segment-limit.
• If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
• If offset is legal, then it is added to the segment-base to produce the physical-memory address.

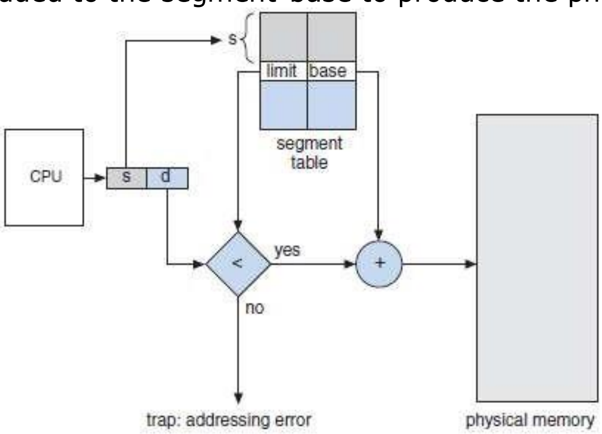


Figure 3.27 Segmentation hardware