4.1 DEADLOCKS

- When processes request a resource and if the resources are not available at that time the process enters into waiting state. Waiting process may not change its state because the resources they are requested are held by other process. This situation is called deadlock.
- The situation where the processes wait for each other to realease the resource held by another process is called deadlock.

4.2 SYSTEM MODEL

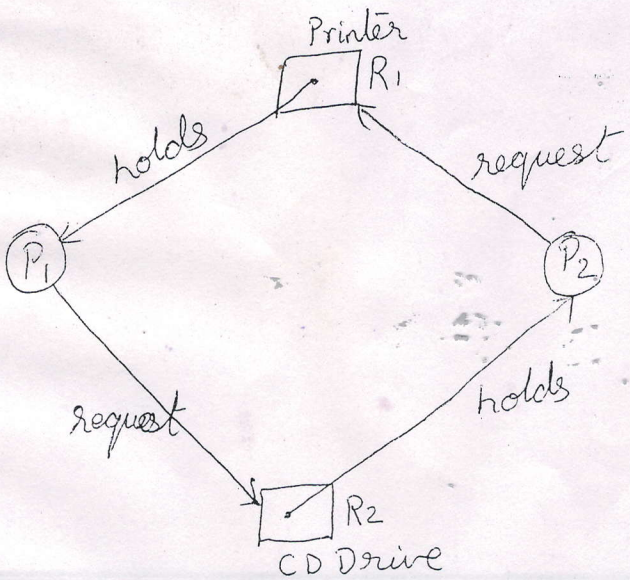
- A system consists of finite number of resources and is distributed among number of processes.
- A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

A process may utilize the resources in only the following sequences:

1. Request:-If the request is not granted immediately then the requesting process must wait until it can acquire the resources.
2. Use:-The process can operate on the resource.
3. Release:-The process releases the resource after using it.

Deadlock may involve different types of resources.

For eg:-Consider a system with one printer (R1) and one tape drive (R2). If a process Pi currently holds a printer R1 and a process Pj holds the tape drive R2. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs.



4.3 DEADLOCK CHARACTERIZATION

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-

1. **Mutual Exclusion**: Only one process is holding the resource at a time. If any other process requests for the resource, the requesting process must wait until the resource has been released.

2. **Hold and Wait**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.

3. **No Preemption**: Resources cannot be preempted i.e., only the process holding the resources must release it after the process has completed its task.

4. **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 resource held by process Pn and Pn is waiting for the resource held by P1.

All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:

Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).
The set of vertices (v) can be described into two different types of nodes

1. P={P1,P2.....Pn} i.e., set consisting of all active processes, represented by circle.
2. R={R1,R2.....Rn} i.e., set consisting of all resource types in the system, represented by rectangle. With dot representing the instances of that resource type.

There are two types of edges :

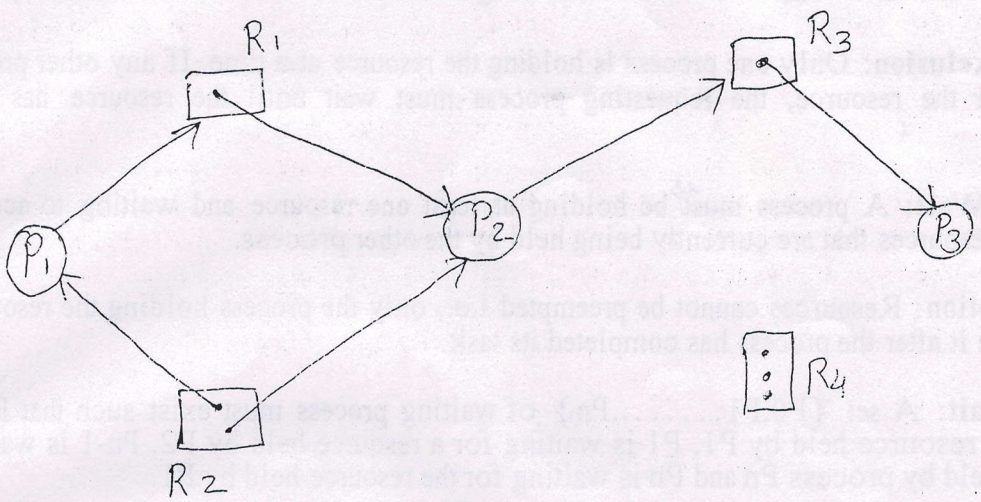
1. A directed edge from process Pi to resource type Rj denoted by Pi→Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called **Request edge**.
2. A directed edge Ri Pj signifies that resource Rj is held by process Pi. This is called **Assignment edge/ Allocation edge.**

The sets P, *R,* and E:

P={P1,P2,P3}
R= {R1,R2,R3,R4}
E= {P1→R1, P2→P3, R1→P2, R2→P2, R2→P1, R3→P3 }

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type *R3*
- Three instances of resource type *R4*



- If the graph contains no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.
- If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted.

4.4 METHODS FOR HANDLING DEADLOCKS

Deadlock problem can be solved in one of three ways:

- Use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state, detect it, and recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

4.5 DEADLOCK PREVENTION

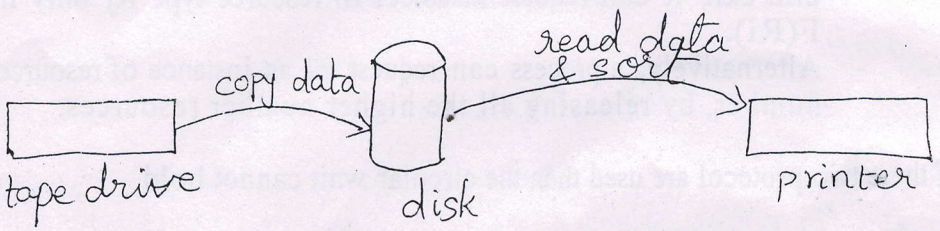
For a deadlock to occur each of the four necessary conditions must hold. If at least one of the conditions does not hold then we can prevent occurrence of deadlock.

1. Mutual Exclusion: This holds for non-sharable resources. *Eg*:-A printer can be used by only one process at a time.
   Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never

waits for accessing in a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

2. Hold and Wait: This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource.
   - Request all its resources before execution begins
     Eg:-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. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

   - Request for resource only when it has none of the resources.
     The process is allocated with tape drive and disk file, first. It performs the required operation and releases both. Then the process once again request for disk file and the printer.



3. NoPreemption: To ensure that this condition never occurs the resources must be preempted. The following protocol can be used.
   - If a process is holding some resource(R1,R2,R3) and request another resource(R4) that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.
   - When a process (P1) request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process(P2). If so we preempt the resources from the waiting process(P2) and allocate them to the requesting process(P1). If resource is not available with P2, P1 has to wait.

4. Circular Wait:-The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never occurs, is to impose ordering on all resource types and each process requests resource in an increasing order.
   -Order all resource types as per number of instances of each type.
   -Process should request resource in an increasing order of enumeration

   Let R={R1,R2,.....Rn} be the set of resource types. We assign each resource type, with a unique integer value. This will allows us to compare two

resources and determine whether one precedes the other in ordering.

Eg:-we can define a one to one function

F:R→N as follows : where the value 'N' indicates the no. of instances of the resource.

F(tape drive)=1
F(disk drive)=5
F(printer)=12

A process holding 'disk drive', cannot request for tapedrive. but it can request for printer.

Deadlock can be prevented by using the following protocol:

- Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri initially and then, it can request instances of resource type Rj only if F(Rj) > F(Ri).
- Alternatively, a process can request for an instance of resource of lower number, by releasing all the higher number resources.

If these two protocol are used then the circular wait cannot hold.

Eg; If a process is having an instance of disk drive, whose 'N' value is set as '5'.

- Then it can request for printer only. Because F(Printer)>F(disk drive).

- If the process wants the tape drive, it has to release all the resources held.

4.6 DEADLOCK AVOIDANCE

Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.

- Avoiding deadlocks requires additional information about the sequence in which the resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- For each requests it requires to check whether the resources is currently available. If resources is available, the OS checks if the resource is assigned will the system lead to deadlock. If no then the actual allocation is done.

Safe State:

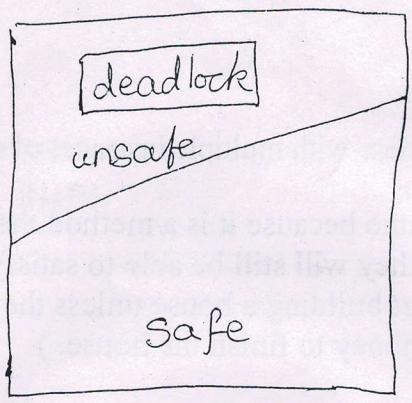
A safe state is a state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.

A system is in safe state if there exists a safe sequence.

A sequence of processes <P1,P2,.....Pn> is a safe sequence for the current allocation state

if for each Pi process request can be satisfied by the currently available resources.

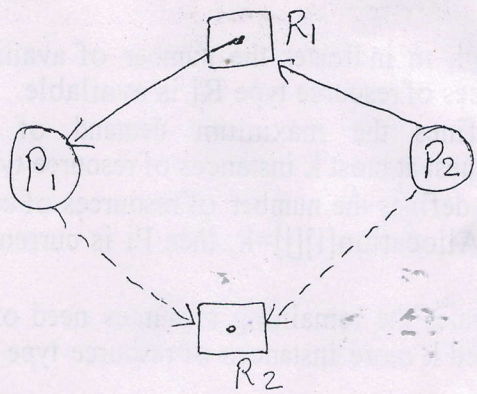
- If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- A safe state is not a deadlock state.
- Whenever a process request a resource i.e., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.



The system in unsafe state may lead to deadlock.

Resource Allocation Graph Algorithm:

This algorithm is used only if we have one instance of a resource type. In addition to the request edge and the assignment edge a new edge called claimedge is used. *For eg*:-A claim edge Pi---- >Rj indicates that process Pi may request Rj in future. The claim edge is represented by a dotted line.



Process P_1 & P_2 has claim edge to R_2 $\Rightarrow$ P_1 & P_2 may request for R_2 in future

When a process Pi requests the resource Rj, the claim edge is converted to a request edge.
When resource Rj is released by process Pi, the assignment edge Rj→Pi is replaced by the

claim edge Pi----- >Rj.

When a process Pi requests resource Rj the request is granted only if converting the request edge Pi→Rj to as assignment edge Rj-→Pi do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state.

Banker's Algorithm:

This algorithm is applicable to the system with multiple instances of each resource types.

The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)

When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.

When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

Several data structures are used to implement the banker's algorithm.
Let 'n' be the number of processes in the system
and 'm' be the number of resources types.

We need the following data structures:

- **Available**:-A vector of length m indicates the number of available resources. If Available[i]=k, then k instances of resource type Rj is available.
- **Max**:-An n*m matrix defines the maximum demand of each process if Max[i][j]=k, then Pi may request at most k instances 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]=k, then Pi is currently k instances of resource type Rj.
- **Need**:-An n*m matrix indicates the remaining resources need of each process. If Need[i][j]=k, then Pi may need k more instances of resource type Rj to compute its task.

So Need[i][j]=Max[i][j] – Allocation[i]

Safety Algorithm:

This algorithm is used to find out whether or not a system is in safe state or not.

Step 1. Let work and finish be two vectors of length m and n respectively.
Initialize work = available and Finish[i]=false for i=1,2,3,.....n

Step 2. Find i such that both Finish[i]=false and Need i <= work
If no such i exist then go to step 4

Step 3. Work = work + Allocation Finish[i]=true Go to step 2

Step 4. If finish[i]=true for all i, then the system is in safe state. This algorithm may require an order of m*n*n operation to decide whether a state is safe.

Resource Request Algorithm:

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. When a request for resources is made by process Pi the following actions are taken.

1) If Request(i) <= Need(i) go to step 2
otherwise raise an error condition since the process has exceeded its maximum claim.

2) If Request(i) <= Available go to step 3
otherwise Pi must wait. Since the resources are not available.

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)

If the resulting resource allocation state is safe[for this, safety algorithm has to be checked], the transaction is complete and Pi is allocated its resources. If the new state is unsafe then Pi must wait for $Request_i$ and old resource allocation state is restored.

Consider the following snapshot of a system.

Process	Allocation			Maximum			Available			← freely available R in system
	A	B	C	A	B	C	A	B	C	
P_0	0	1	0	7	5	3	3	3	2	
P_1	2	0	0	3	2	2				
P_2	3	0	2	9	0	2				
P_3	2	1	1	2	2	2				
P_4	0	0	2	4	3	3				

Answer the following questions using Bankers Algorithm.

i) Is the system in a safe state?

ii) If a request from P_1 arrives for (1, 0, 2) can the request be granted immediately?

Total instances of resources in system -

A = 3+7 = 10 , B = 3+2 = 5 , C = 2+5 = 7

Need Matrix

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

i) Safety Algorithm

Work = [3, 3, 2] , Finish [P_0 - P_4] = false.

1) i = P_0; Need[P_0] ≤ Work

[7, 4, 3] ≤ [3, 3, 2] ✗

P_0 cannot execute now.

2) i = P_1, Need[P_1] ≤ Work

[1, 2, 2] ≤ [3, 3, 2]

P_1 executes...

Work = Work + Allocation[P_1].

= [3, 3, 2] + [2, 0, 0] = [5, 3, 2]

Finish [P_1] = true

3) i = P_2, Need[P_2] ≤ Work

[6, 0, 0] ≤ [5, 3, 2] ✗

P_2 cannot execute now.

4) i = P_3, Need[P_3] ≤ Work.

[0, 1, 1] ≤ [5, 3, 2] ✓

P_3 executes → Work = Work + Allocation[P_3]

= [5, 3, 2] + [2, 1, 1] = [7, 4, 3] Finish[P_3] = true

5) $i = P_4$, $Need[P_4] \leq Work$

$[4,3,1] \leq [7,4,3]$

P_4 executes

$Work = Work + Allocation[P_4]$

$= [7,4,3] + [0,0,2]$

$= [7,4,5]$.

$Finish[P_4] = true$.

6) $i = P_0$, $Need[P_0] \leq Work$

$[7,4,3] \leq [7,4,5]$

P_0 executes

$Work = Work + Allocation[P_0]$

$= [7,4,5] + [0,1,0] = [7,5,5]$

$Finish[P_0] = true$.

7) $i = P_2$, $Need[P_2] \leq Work$

$[6,0,0] \leq [7,5,5]$

P_2 executes.

$work = Work + Allocation[P_2]$

$= [7,5,5] + [3,0,2] = [10,5,7]$

$Finish[P_2] = true$.

$Finish[P_0 - P_4] = true$, All the processes in the system can be executed. So the system is in safe state

The safe sequence is $<P_1, P_3, P_4, P_0, P_2>$

ii) P_1 request $\rightarrow [1,0,2]$

1) $Request_{[P_1]} \leq Need_{[P_1]}$

$[1,0,2] \leq [1,2,2]$ ✓

Requested resource is needed.

2) $Request_{[P_1]} \leq Available$

$[1,0,2] \leq [3,3,2]$

Requested resource is available.

3) Assume that resources is allocated to P_1, then the changes are -

$Allocation_{[P_1]} = Allocation_{[P_1]} + Request_{[P_1]}$

$Allocation_{[P_1]} = [2,0,0] + [1,0,2] = [3,0,2]$

$Need_{[P_1]} = Need_{[P_1]} - Request_{[P_1]}$

$Need_{[P_1]} = [1,2,2] - [1,0,2] = [0,2,0]$

$Available = Available - Request_{[P_1]}$

$= [3,3,2] - [1,0,2] = [2,3,0]$

Now, with these changes safety algorithm is checked to find if the system is in safe state.

	Allocation			Maximum			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	3	0	2	3	2	2	0	2	0
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

Available $[2,3,0]$

Work = [2, 3, 0] Finish [P_0 - P_4] = false

1) i = P_0, Need[P_0] ≤ Work

[7, 4, 3] ≤ [2, 3, 0] ✗

P_0 cannot execute now.

2) i = P_1, Need[P_1] ≤ Work

[0, 2, 0] ≤ [2, 3, 0] ✓

P_1 executes.

Work = Work + Allocation[P_1]

= [2, 3, 0] + [3, 0, 2] = [5, 3, 2]

Finish[P_1] = true

3) i = P_2, Need[P_2] ≤ Work

[6, 0, 0] ≤ [5, 3, 2] ✗

P_2 cannot execute now.

4) i = P_3, Need[P_3] ≤ Work

[0, 1, 1] ≤ [5, 3, 2] ✓

P_3 executes.

Work = Work + Allocation[P_3]

= [5, 3, 2] + [2, 1, 1] = [7, 4, 3]

Finish[P_3] = true.

5) i = P_4, Need[P_4] ≤ Work

[4, 3, 1] ≤ [7, 4, 3] ✓

P_4 executes

Work = Work + Allocation[P_4]

= [7, 4, 3] + [0, 0, 2] = [7, 4, 5].

Finish[P_4] = true

6) i = P_0, Need[P_0] ≤ Work

[7, 4, 3] ≤ [7, 4, 5] ✓

Work = [7, 4, 5] + [0, 1, 0] = [7, 5, 5]

Finish[P_0] = true.

Resource Request Algorithm:

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. When a request for resources is made by process Pi the following actions are taken.

1) If Request(i) <= Need(i) go to step 2
otherwise raise an error condition since the process has exceeded its maximum claim.

2) If Request(i) <= Available go to step 3
otherwise Pi must wait. Since the resources are not available.

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)

If the resulting resource allocation state is safe[for this, safety algorithm has to be checked], the transaction is complete and Pi is allocated its resources. If the new state is unsafe then Pi must wait for Request$_i$ and old resource allocation state is restored.

7) $i = P_2$,

Need[P_2] ≤ Work

[6,0,0] ≤ [7,5,5]

Work = Work + Allocation[P_2]

= [7,5,5] + [3,0,2]

= [10,5,7]

Finish[P_2] = true

As all Finish [P_0-P_4] = true. ~~Even if requested~~

All the processes have executed completely.

The system is in safe state.

The safe sequence is <P_1, P_3, P_4, P_0, P_2>

4.7 DEADLOCK DETECTION

Another way of handling deadlock is, allow the deadlock to occur, then detect and recover from deadlock.

In this environment the system may provide

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

Single Instances of each Resource Type:

If all the resources have only a single instance then we can define deadlock detection algorithm that uses a variant of resource allocation graph called a **wait for graph**. This graph is obtained by removing the nodes of type resources and removing appropriate edges.

An edge from Pi to Pj in wait for graph implies that Pi is waiting for Pj to release a resource that Pi needs.

An edge from Pi to Pj exists in wait for graph if and only if the corresponding resource allocation graph contains the edges Pi Rq and Rq Pj.

Deadlock exists within the system if and only if there is a cycle.

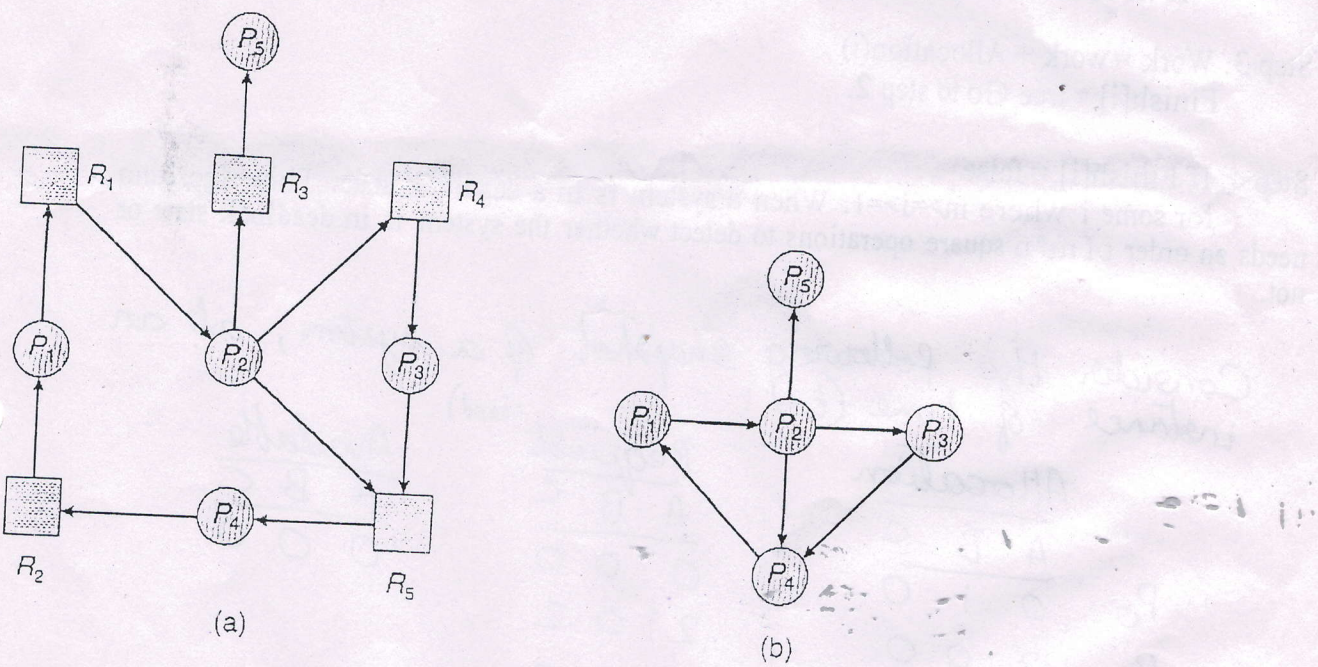


Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Several Instances of a Resource Types:

The wait for graph is applicable to only a single instance of a resource type. The following algorithm applies if there are several instances of a resource type. The following data structures are used:-

- Available:-Is a vector of length m indicating the number of available resources of each type .
- Allocation:-Is an m*n matrix which defines the number of resources of each type currently allocated to each process.
- Request:-Is an m*n matrix indicating the current request of each process. If request[i,j]=k then Pi is requesting k more instances of resources type Rj.

Step 1. Let work and finish be vectors of length m and n respectively.
Initialize Work = available
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 Finish[i]= false and Request(i)<=work
If no such i exist go to step 4.

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

Step 4. If Finish[i] = false
for some i where m>=i>=1. When a system is in a deadlock state. This algorithm needs an order of m*n square operations to detect whether the system is in deadlock state or not.

Consider the following snapshot of a system, at an instance of time (t0):

	Allocation			Request (Need)			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

If the requested resources are allocated, check whether system will be in safe state, Find the safe sequence.

Work = (0,0,0), Finish [P_0 – P_4] = false ; as all processes are allocated with resources.

1) i = P_0 ; Request$_i$ ≤ Work.

(0,0,0) ≤ (0,0,0)

Work = Work + Allocation$_i$

= (0,0,0) + (0,1,0) = (0,1,0)

Finish[P_0] = true.

2) i = P_2, (0,0,0) < (0,1,0)

Work = (0,1,0) + (3,0,3) = (3,1,3)

Finish[P_2] = true

3) i = P_3, (1,0,0) < (3,1,3)

Work = (3,1,3) + (2,1,1) = (5,2,4)

Finish[P_3] = true.

4) i = P_4, (0,0,2) < (5,2,4)

Work = (5,2,4) + (0,0,2) = (5,2,6)

Finish[P_4] = true

5) i = P_1, (2,0,2) < (5,2,6)

Work = (5,2,6) + (2,0,0) = (7,2,6)

Finish[P_1] = true

As Finish[P_0 – P_4] = true, system is in safe state when the requested resources are allocated.

The safe sequence is < P_0, P_2, P_3, P_4, P_1 >

~~[Suppose if the request of P_2 is given as (0,0,1) check the sequence in Page 17]~~

7.8 Recovery From Deadlock

- There are two basic approaches to recovery from deadlock:
  1. Process Termination
  2. Resource Preemption

7.8.1 Process Termination

- Two basic approaches to terminate processes:
  - **Abort all deadlocked processes.** This method breaks the deadlock cycle by terminating all the processes involved in deadlock. But, at a great expense, the deadlocked 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.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- In the latter case there are many factors that can go into deciding which processes to terminate next:
  1. Process priorities.
  2. How long the process has been running, and how close it is to finishing.
  3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
  4. How many more resources does the process need to complete.
  5. How many processes will need to be terminated
  6. Whether the process is interactive or batch.
  7. (Whether or not the process has made non-restorable changes to any resource.)

7.7.2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
  1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)
  3. **Starvation** – There are chances that the same resource is picked from processes as a victim, every time the deadlock occurs and this continues. This is starvation. A count can be kept on number of rollback of a process and the process has to be a victim for finite number of times only.

Modulue III

Main Memory Management Strategies

- Every program to be executed has to be executed must be in memory. The instruction must be fetched from memory before it is executed.
- In multi-tasking OS memory management is complex, because as processes are swapped in and out of the CPU, their code and data must be swapped in and out of memory.

Basic Hardware

Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can access directly.

The program and data must be bought into the memory from the disk, for the process to run. Each process has a separate memory space and must access only this range of legal addresses. Protection of memory is required to ensure correct operation. This prevention is provided by hardware implementation. Two registers are used - a base register and a limit register. The base register holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

For example, if the base register holds 1000 and limit register is 500, then the program can legally access all addresses from 1000 through 1500 (inclusive).

Protection of memory space is done. Any attempt by an executing program to access operating-system memory or other program memory results in a trap to the operating system, which treats the attempt as a fatal error. This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode only the operating system can load the base and limit registers.

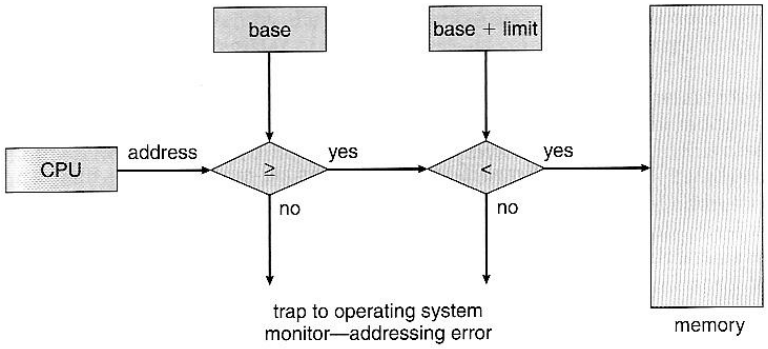


Figure 8.2 Hardware address protection with base and limit registers.

Address Binding

- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:
  - **Compile Time** - If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled.
  - **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  - **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.
- Figure 8.3 shows the various stages of the binding processes and the units involved in each stage:

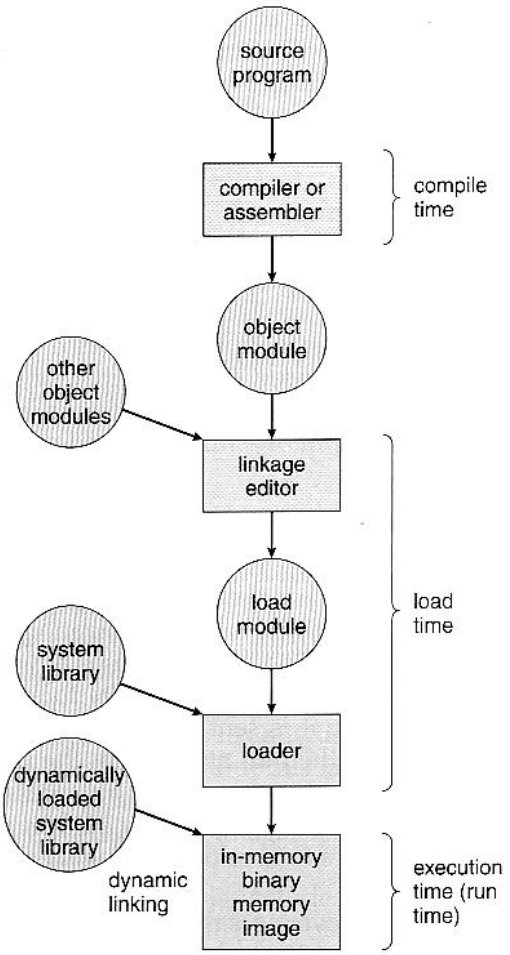


Figure 8.3 Multistep processing of a user program.

Logical Versus Physical Address Space

- The address generated by the CPU is a ***logical address***, whereas the memory address where programs are actually stored is a ***physical address***.
- The set of all logical addresses used by a program composes the ***logical address space***, and the set of all corresponding physical addresses composes the ***physical address space.***
- The run time mapping of logical to physical addresses is handled by the ***memory-management unit, MMU***.
    - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
    - The base register is now termed a ***relocation register***, whose value is added to every memory request at the hardware level.

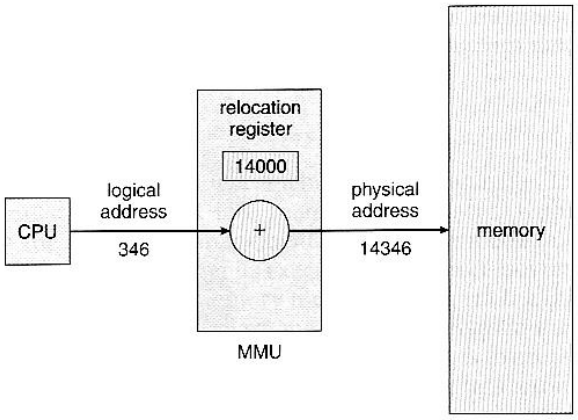


Figure 8.4 Dynamic relocation using a relocation register.

8.1.4 Dynamic Loading

- Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need not be loaded, thus reducing total memory usage and generating faster program startup times. The disadvantage is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

8.1.5 Dynamic Linking and Shared Libraries

- With ***static linking*** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With ***dynamic linking***, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

- This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
- An added benefit of ***dynamically linked libraries*** (DLLs, also known as ***shared libraries*** or ***shared objects*** on UNIX systems) involves easy upgrades and updates.

8.2 Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes that are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***
- Swapping is **the process of moving a process from memory to backing store and moving another process from backing store to memory**. Swapping is a very slow process compared to other operations.
- It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again. Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.

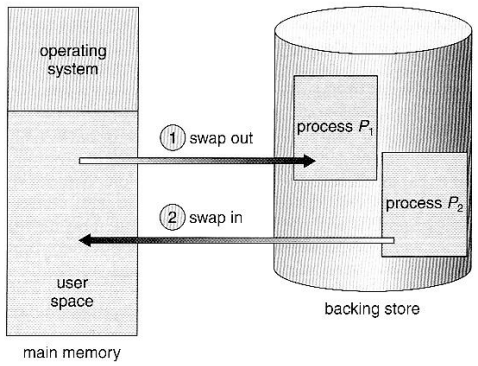


Figure 8.5 Swapping of two processes using a disk as a backing store.

8.3 Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is

where the interrupt vectors are located). Here each process is contained in a single contiguous section of memory.

8.3.1 Memory Mapping and Protection

- The system shown in figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

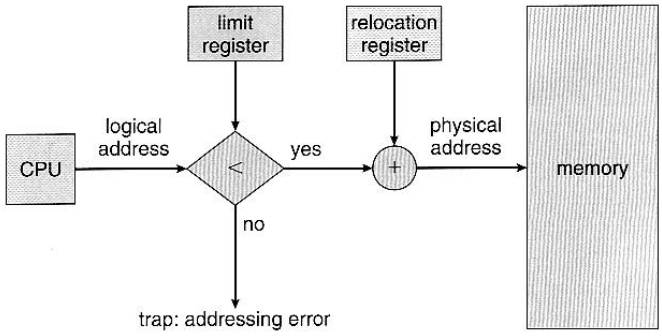


Figure 8.6 Hardware support for relocation and limit registers.

8.3.2 Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into **equal sized** partitions, and to assign each process to their own partition (called as **MFT**). This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a **suitable size** whenever a process needs to be loaded into memory (called as **MVT**). There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:
    1. **First fit** - Search the list of holes until one is found that is **big enough** to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
    2. **Best fit** - Allocate the ***smallest*** **hole that is big enough** to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
    3. **Worst fit** - Allocate the **largest hole** available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.
- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

8.3.3. Fragmentation

The allocation of memory to process leads to fragmentation of memory. A hole is the free space available within memory. The two types of fragmentation are –

- External fragmentation – holes present in between the process
- Internal fragmentation - holes are present within the process itself. ie. There is free space within a process.

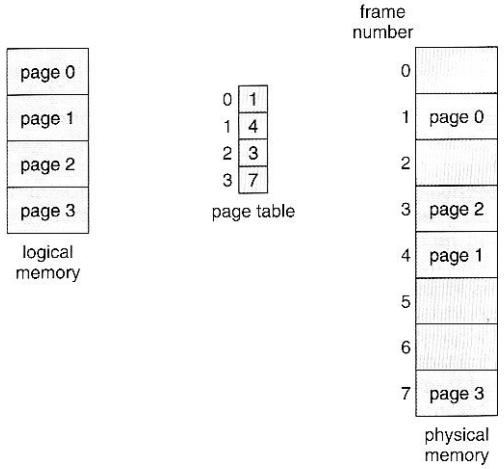
- ***Internal fragmentation*** occurs with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size.
- If the programs in memory are relocatable, (using execution-time address binding), then the **external fragmentation** problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory so as to place all free memory together to get a large free block. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution to external fragmentation is to allow processes to use non-contiguous blocks of physical memory- **Paging** and **Segmentation**.

8.4 Paging

- Paging is a memory management scheme that allows processes to be stored in physical memory discontinuously. It eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

8.4.1 Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a program's logical memory space into blocks of the same size called ***pages.***



- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up which frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory.

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the

maximum size of each page, and should correspond to the system frame size.)

- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

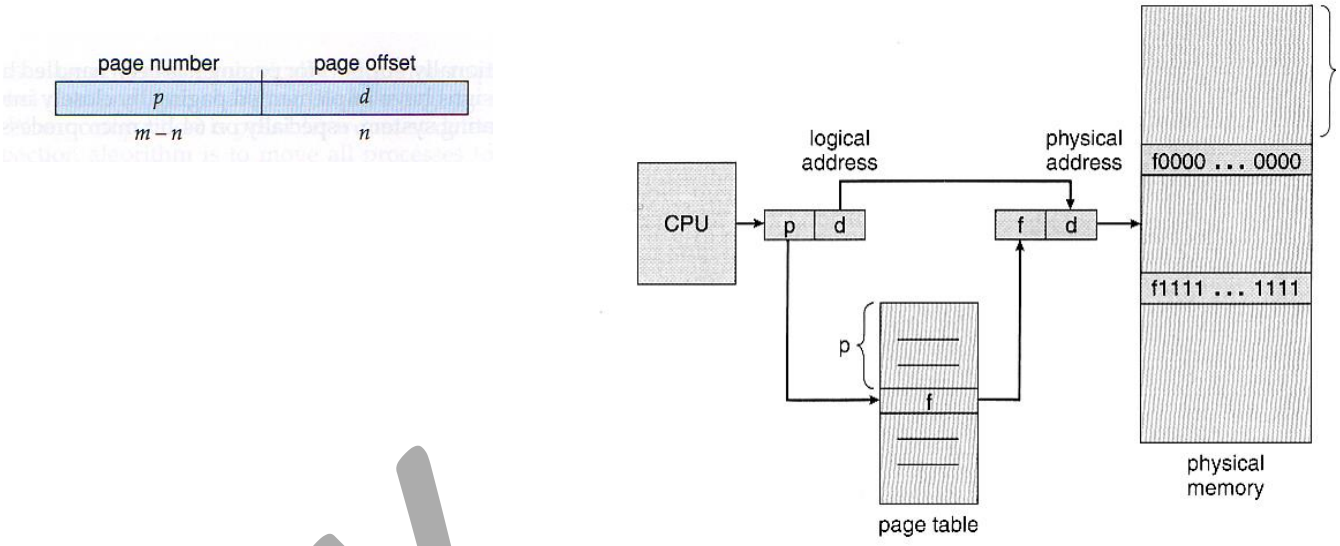


Figure 8.7 Paging hardware.

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is **no external fragmentation** with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.

- Consider the following example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

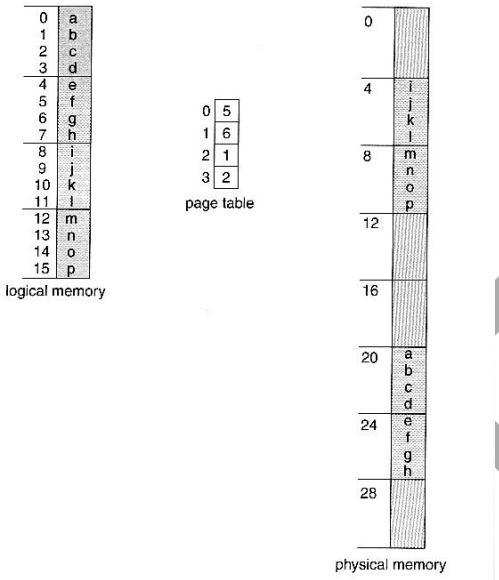


Figure 8.9 Paging example for a 32-byte memory with 4-byte pages.

- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

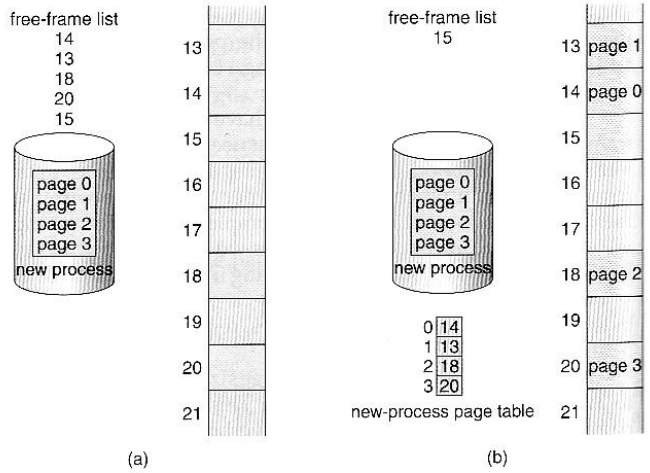
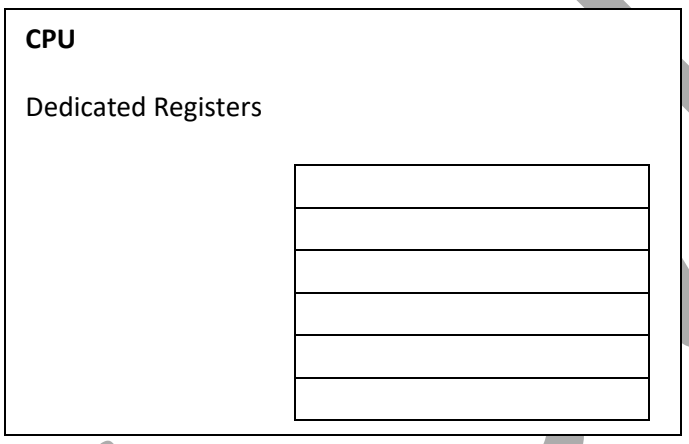


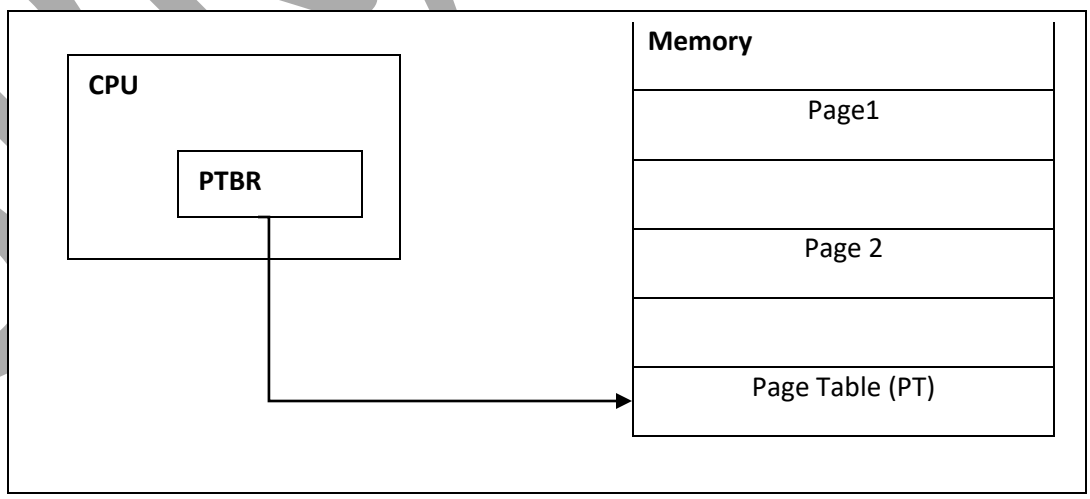
Figure 8.10 Free frames (a) before allocation and (b) after allocation.

8.4.2 Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use **a set of dedicated registers** for the page table. Here each register content is loaded, when the program is loaded into memory. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)



- An alternate option is to store the page table in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record the address of the page table is memory.
  - Process switching is fast, because only the single register needs to be changed.
  - However memory access is slow, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.



- The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***
  - The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.
  - It is used as a cache device.
  - Addresses are first checked against the TLB, and if the page is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
  - If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
  - Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically these would be kernel frames.

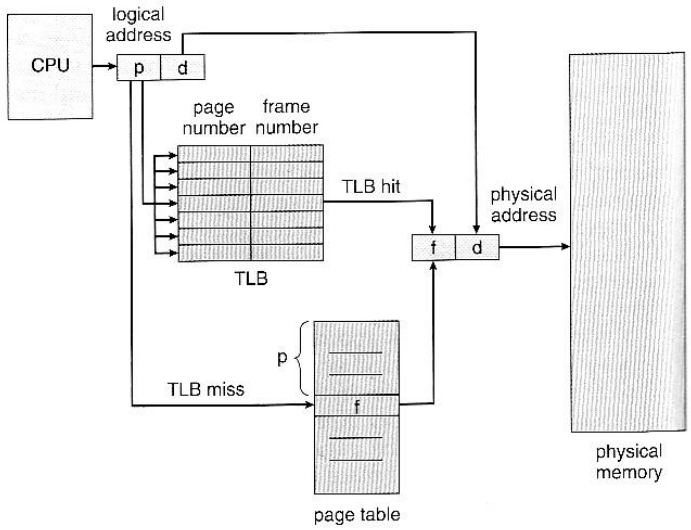


Figure 8.11 Paging hardware with TLB.

- Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.
- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

  0.80 * 120 + 0.20 * 220 = 140 nanoseconds

  for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

8.4.3 Protection

- The page table can also help to protect processes from accessing memory.
- A bit can be added to the page table. Valid / invalid bits can be added to the page table. The valid bit 'V' shows that the page is valid and updated, and the invalid bit 'i' shows that the page is not valid and updated page is not in the physical memory.
- Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. Many processes do not use all the page table entries available to them.
- Addresses of the pages 0,1,2,3,4 and 5 are mapped using the page table as they are valid.
- Addresses of the pages 6 and 7 are invalid and cannot be mapped. Any attempt to access those pages will send a trap to the OS.

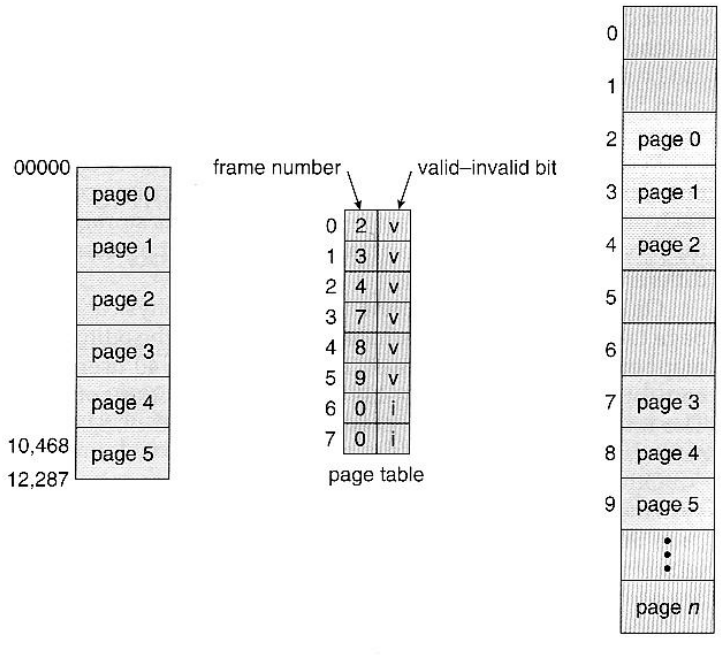


Figure 8.12 Valid (v) or invalid (i) bit in a page table.

8.4.4 Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- If code is ***reentrant***(read-only files) that means that it does not write to or change the code in any way. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.
- Some systems also implement shared memory in this fashion.

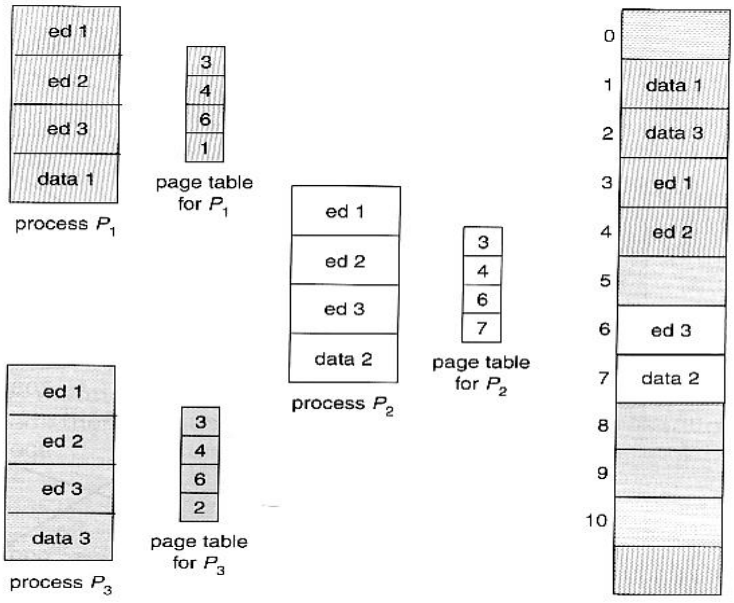


Figure 8.13 Sharing of code in a paging environment.

8.5 Structure of the Page Table

8.5.1 Hierarchical Paging

- This structure supports two or more page tables at different levels (tiers).
- Most modern computer systems support logical address spaces of 2^32 to 2^64.

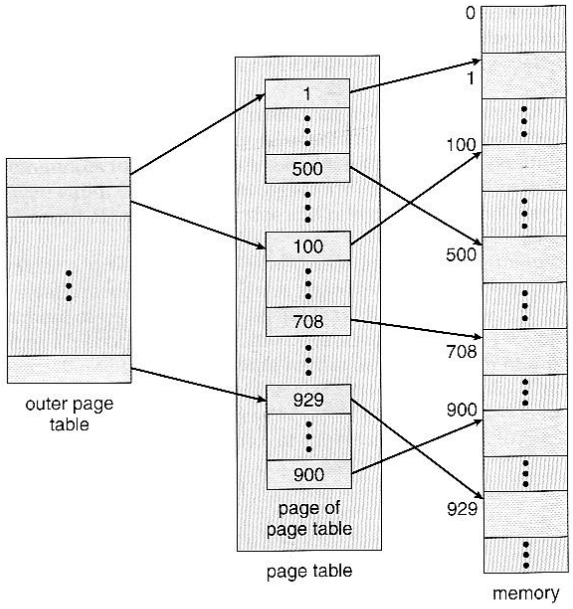
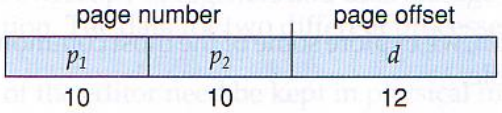
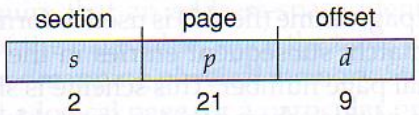


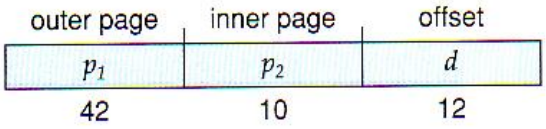
Figure 8.14 A two-level page-table scheme.

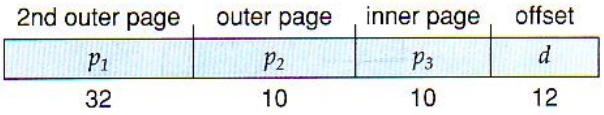


- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form of:



- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging. One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively **slow memory access**. So some other approach must be used.





8.5.2 Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***. Figure 8.16 below illustrates a ***hashed page table*** using chain-and-bucket hashing:

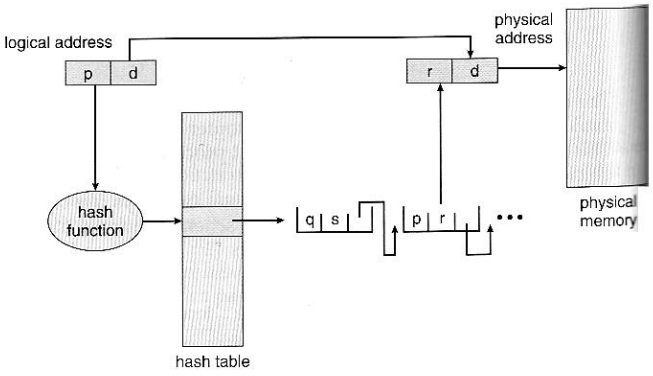


Figure 8.16 Hashed page table.

8.5.3 Inverted Page Tables

- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (i.e. there is one entry per ***frame*** instead of one entry per ***page***.)
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page .
- The 'id' of process running in each frame and its corresponding page number is stored in the page table.

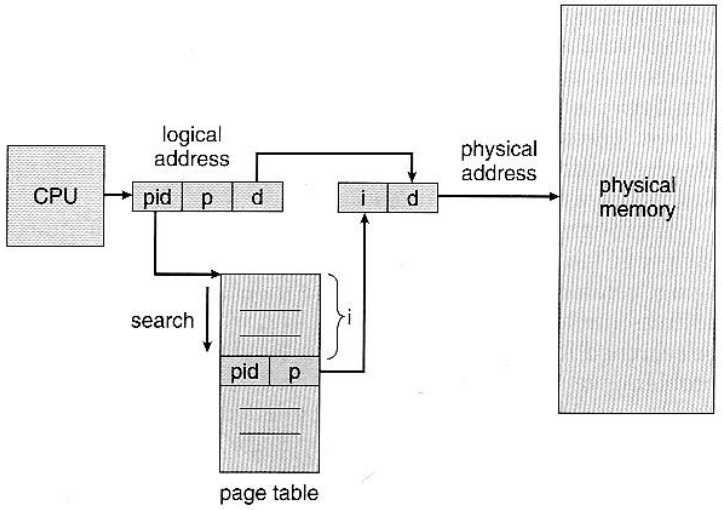


Figure 8.17 Inverted page table.

8.6 Segmentation

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- The logical address consists of 2 tuples:

  <segment-number, offset>

- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure 8.18:

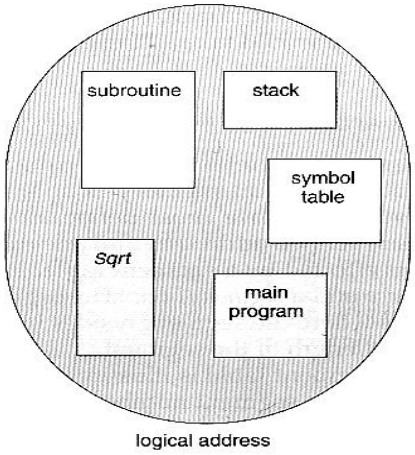


Figure 8.18 User's view of a program.

8.6.2 Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses.
- Each entry in the segment table has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
- A logical address consists of two parts: a **segment number**, *s*, and an **offset** into that segment, *d.* The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base and limit register pairs.

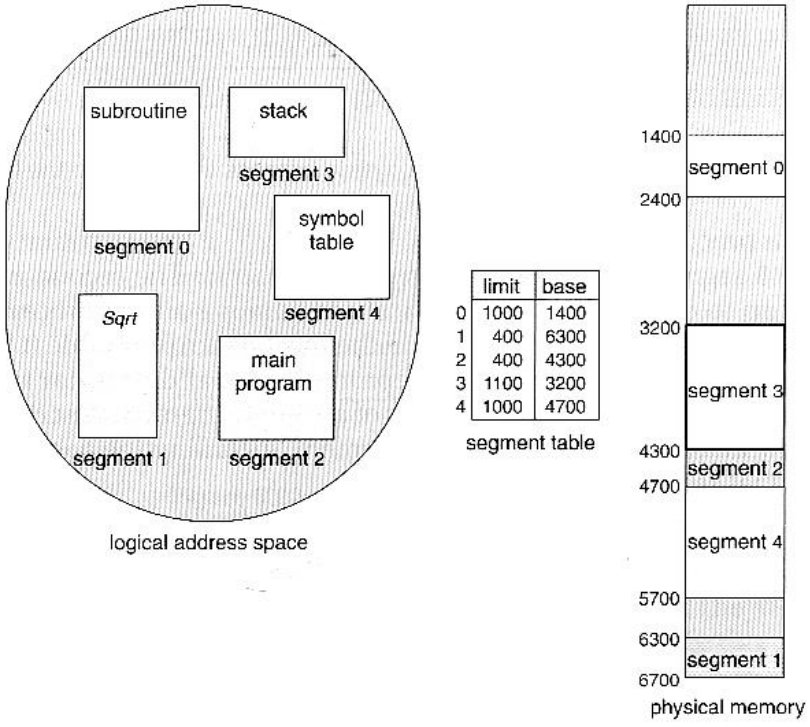


Figure 8.20 Example of segmentation.