## UNIT-3 DEADLOCKS

**System model:**

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, I/O devices are examples of resource types. If a system has 2 CPUs, then the resource type CPU has 2 instances.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its task. The number of resources as it requires to carry out its task. The number of resources requested may not exceed the total number of resources available in the system. A process cannot request 3 printers if the system has only two.

A process may utilize a resource in the following sequence:

* REQUEST: The process requests the resource. If the request cannot be granted immediately (if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
* USE: The process can operate on the resource .if the resource is a printer, the process can print on the printer.
* RELEASE: The process release the resource.

For each use of a kernel managed by a process the operating system checks that the process has requested and has been allocated the resource. A system table records whether each resource is free (or) allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

To illustrate a deadlocked state, consider a system with 3 CDRW drives. Each of 3 processes holds one of these CDRW drives. If each process now requests another drive, the 3 processes will be in a deadlocked state. Each is waiting for the event “CDRW is released” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. Consider a system with one printer and one DVD drive. The process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

## 

## DEADLOCK CHARACTERIZATION:

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

## NECESSARY CONDITIONS:

A deadlock situation can arise if the following 4 conditions hold simultaneously in a system:

1. MUTUAL EXCLUSION: Only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed 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 otherprocesses.
3. NO PREEMPTION: Resources cannot be preempted. A resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. CIRCULAR WAIT: A set {P0,P1,…..Pn} of waiting processes must exist such that P0 is waiting for resource held by P1, P1 is waiting for a resource held by P2,……,Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by P0.

## 

## RESOURCE ALLOCATION GRAPH

Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph. This graph consists of a set of vertices V and a set of edges E. the set of vertices V is partitioned into 2 different types of nodes:

* P = {P1, P2….Pn}, the set consisting of all the active processes in the system. R= {R1, R2….Rm}, the set consisting of all resource types in the system.
* A directed edge from process Pi to resource type Rj is denoted by Pi ->Rj. It signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
* A directed edge from resource type Rj to process Pi is denoted by Rj ->Pi, it signifies that an instance of resource type Rj has been allocated to process Pi.
* A directed edge Pi ->Rj is called a requested edge. A directed edge Rj->Piis called an assignment edge.

We represent each process Pi as a circle, each resource type Rj as a rectangle. Since resource type Rj may have more than one instance. We represent each such instance as a dot within the rectangle. A request edge points to only the rectangle Rj. An assignment edge must also designate one of the dots in the rectangle.

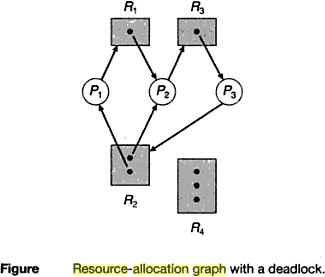
When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource, as a result, the assignment edge is deleted.

The sets P, R, E:

P= {P1, P2, P3}

R= {R1, R2, R3, R4}

E= {P1 ->R1, P2 ->R3, R1 ->P2, R2 ->P2, R2 ->P1, R3 ->P3}



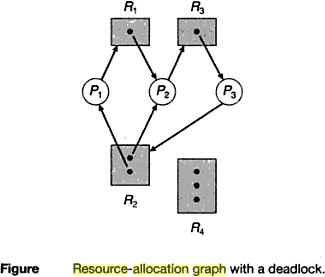
* One instance of resource type R1
* Two instances of resource type R2
* One instance of resource type R3
* Three instances of resource type R4

**PROCESS STATES:**

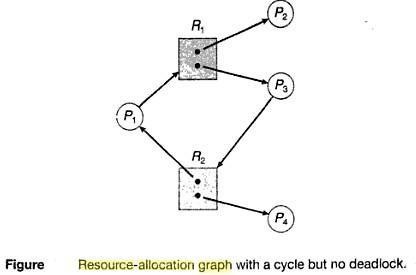
* Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
* Process P2 is holding an instance of R1 and an instance of R2 and is waiting for instance of R3. Process P3 is holding an instance of R3.
* If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
* Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 ->R2 is added to the graph.

2 cycles:

P1 ->R1 ->P2 ->R3 ->P3 ->R2 ->P1 P2 ->R3 ->P3 ->R2 ->P2



Processes P1, P2, P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3.process P3 is waiting for either process P1 (or) P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.



We also have a cycle: P1 ->R1 ->P3 ->R2 ->P1

However there is no deadlock. Process P4 may release its instance of resource type R2.

That resource can then be allocated to P3, breaking the cycle.

## 

## DEADLOCK PREVENTION

For a deadlock to occur, each of the 4 necessary conditions must held By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
2. **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

* Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
* Low resource utilization; starvation possible

## No Preemption –

* If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
* Preempted resources are added to the list of resources for which the process is waiting
* Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

1. **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

**Deadlock Avoidance**

Requires that the system has some additional *a priori* information available

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

## 

## Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

System is in **safe state** if there exists a sequence <*P1, P2, …, Pn*> of ALL the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j* <*I*

That is:

* If Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* have finished
* When *Pj* is finished, *Pi* can obtain needed resources, execute, return allocated resources, and terminate
* When *Pi* terminates, *Pi* +1 can obtain its needed resources, and so on If a system is in safe state no deadlocks
* If a system is in unsafe state possibility of deadlock Avoidance ensure that a system will never enter an unsafe state.

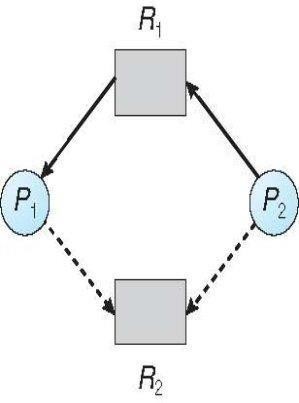
**Avoidance algorithms**

Single instance of a resource type

* Use a resource-allocation graph Multiple instances of a resource type
* Use the banker’s algorithm

## Resource-Allocation Graph Scheme

**Claim edge** *Pi* , *Rj* indicated that process *Pj* may request resource *Rj*; represented by a dashed line. Claim edge converts to request edge when a process requests a resource Request edge converted to an assignment edge when the resource is allocated to the process When a resource is released by a process, assignment edge reconverts to a claim edge Resources must be claimed *a priori* in the system



## Unsafe State In Resource-Allocation Graph

**Banker’s Algorithm**

Multiple instances Each process must a priori claim maximum use When a process requests a resource it may have to wait When a process gets all its resources it must return them in a finite amount of time Let *n* = number of processes, and *m* = number of resources types.

**Available***:* Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type *Rj* available

**Max***: n x m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*

**Allocation***: n* x *m* matrix. If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj*

**Need***: n* x *m* matrix. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj* to complete its task

*Need* [*i,j]* = *Max*[*i,j*] – *Allocation* [*i,j*]

## 

## Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.

Initialize: Work = Available

Finish [i] = false for i = 0, 1, …,n- 1

1. Find an i such that both:
2. Finish [i] = false
3. Need i=Work

If no such i exists, go to step 4

1. Work = Work + Allocationi Finish[i] = true

go to step 2

1. IfFinish [i] == true for all i, then the system is in a safe state

## 

## Resource-Request Algorithm for Process *Pi*

*Request* = request vector for process *Pi*. If *Request i*[*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*

1. If *Request i*£ *Need i* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If *Request i*£ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available
3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

*Available* = *Available* – *Request; Allocation i* = *Allocation i* + *Request i*;

*Need i* = *Need i* – *Request i;*

* *If safe the resources are allocated to Pi*
* *If unsafe Pi must wait, and the old resource-allocation state is restored*

## Example of Banker’s Algorithm(REFER CLASS NOTES)

Consider 5 processes *P*0 through *P*4; 3 resource types:

*A* (10 instances), *B* (5instances), and *C* (7 instances)

Snapshot at time *T*0:

|  |  |  |
| --- | --- | --- |
| *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 2 | 9 0 2 |  |
| *P3-* 2 1 1 | 2 2 2 |  |
| *P4-* 0 0 2 | 4 3 3 |  |

Σ The content of the matrix *Need* is defined to be *Max*

– *Allocation Need A B C*

The system is in a safe state since the sequence <*P*1, *P*3, *P*4, *P*2, *P*0> satisfies safety criteria

## *P*1 Request (1,0,2)

Check that Request £ Available (that is, (1,0,2) £ (3,3,2) true

|  |  |  |
| --- | --- | --- |
| *Allocation* | *Need* | *Available* |
| *A B C* | *A B C* | *A B C* |
| *P*0-0 1 0 | 7 4 3 | 2 3 0 |
| *P*1-3 0 2 | 0 2 0 |  |
| *P*2-3 0 2 | 6 0 0 |  |
| *P*3-2 1 1 | 0 1 1 |  |
| *P*4-0 0 2 | 4 3 1 |  |

Executing safety algorithm shows that sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies safety requirement

## 

## Deadlock Detection

Allow system to enter deadlock state Detection algorithm Recovery scheme

## 

## Single Instance of Each Resource Type

Maintain *wait-for* graph Nodes are processes *Pi* Æ *Pj* if *Pi* is waiting for *Pj* Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

An algorithm to detect a cycle in a graph requires an order of operations, where *n* is the number of vertices in the graph

## Resource-Allocation Graph and Wait-for Graph

Resource-Allocation Graph Corresponding wait-for graph

## Several Instances of a Resource Type

**Available***:* A vector of length *m* indicates the number of available resources of each type. **Allocation***:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

**Request***:* An *n* x *m* 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*.

## 

## Detection Algorithm

Let Work and Finish be vectors of length m and n, respectively Initialize:

1. Work = Available
2. For i = 1,2, …, n, if Allocation iπ 0, then Finish[i] = false; otherwise, Finish[i] = true
3. Find an index i such that both:
4. Finish[i] == false
5. Requesti£Work

If no such i exists, go to step 4

1. *Work* = *Work* + *Allocationi Finish*[*i*] = *true*

go to step 2

1. If *Finish*[*i*] == false, for some *i*, 1 £*i*£*n*, then the system is in deadlock state. Moreover, if

*Finish*[*i*] == *false*, then *Pi* is deadlocked

## 

## Recovery from Deadlock:

**Process Termination**

* Abort all deadlocked processes
* Abort one process at a time until the deadlock cycle is eliminated
* In which order should we choose to abort?
  + Priority of the process
  + How long process has computed, and how much longer to completion
  + Resources the process has used
  + Resources process needs to complete
  + How many processes will need to be terminated
  + Is process interactive or batch?

## 

## Resource Preemption

Selecting a victim – minimize cost

Rollback – return to some safe state, restart process for that state

Starvation – same process may always be picked as victim, include number of rollback in cost factor

## Memory Management

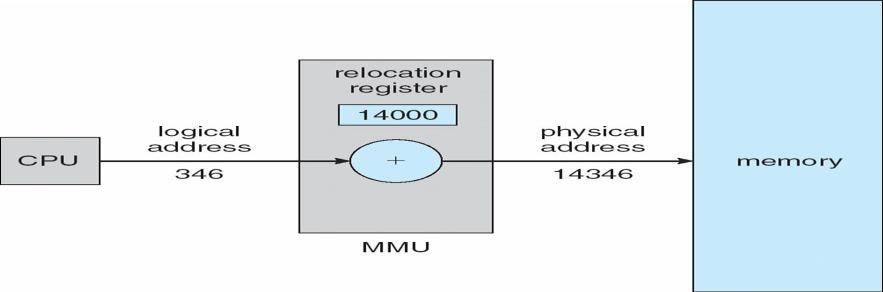
## Logical And Physical Addresses

An address generated by the CPU is commonly refereed as **Logical Address**, whereas the address seen by the memory unit that is one loaded into the memory address register of the memory is commonly refereed as the **Physical Address**. The compile time and load time address binding generates the identical **logical and physical addresses**. However, the execution time addresses binding scheme results in differing **logical and physical addresses**.

The set of all **logical addresses** generated by a program is known as **Logical Address Space**, where as the set of all **physical addresses** corresponding to these logical addresses is **Physical Address Space**. Now, the run time mapping from virtual address to **physical address** is done by a hardware device known as **Memory Management Unit**. Here in the case of mapping the base register is known as **relocation register**. The value in the relocation register is added to the address generated by a user process at the time it is sent to memory

Let's understand this situation with the help of example: If the base register contains the value 1000,then an attempt by the user to address location 0 is dynamically relocated to location 1000,an access to location 346 is mapped to location 1346.

### *Memory-Management Unit (MMU)*

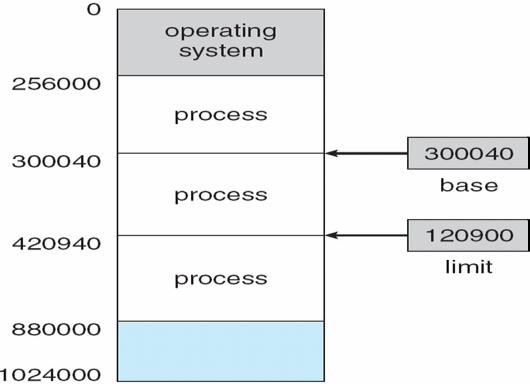
* Hardware device that maps virtual to physical address
* In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
* The user program deals with *logical* addresses; it never sees the *real* physical addresses

The user program never sees the **real physical address** space, it always deals with the **Logical addresses**. As we have two different type of addresses **Logical address** in the range (0 to max) and **Physical addresses** in the range(R to R+max) where R is the value of relocation register. The user generates only **logical addresses** and thinks that the process runs in location to 0 to max. As it is clear from the above text that user program supplies only logical addresses, these **logical addresses** must be mapped to **physical address** before they are used.

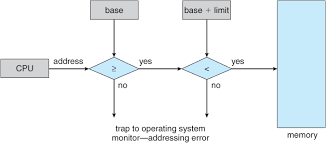
### 

### *Base and Limit Registers*

### A pair of base and limit registers define the logical address space



HARDWARE PROTECTION WITH BASE AND LIMIT



### 

### *Binding of Instructions and Data to Memory*

Address binding of instructions and data to memory addresses can happen at three different stages

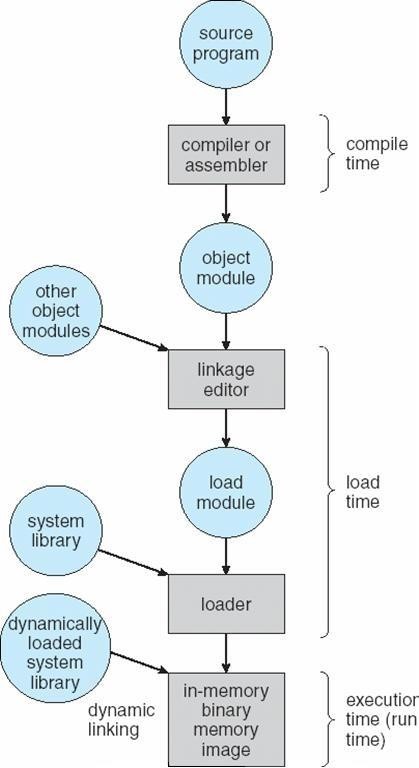
**Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes

**Load time**: Must generate **relocatable code** if memory location is not known at compile time

**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)

### *Multistep Processing of a User Program*



**Dynamic Loading**

* Routine is not loaded until it is called
* Better memory-space utilization; unused routine is never loaded
* Useful when large amounts of code are needed to handle infrequently occurring cases
* No special support from the operating system is required implemented through program design

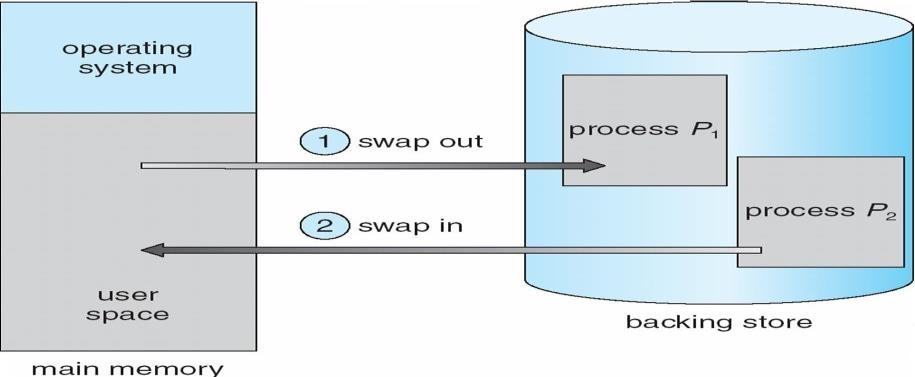
### Dynamic Linking

Linking postponed until execution time Small piece of code, *stub*, used to locate the appropriate memory-resident library routine Stub replaces itself with the address of the routine, and executes the routine Operating system needed to check if routine is in processes’ memory address Dynamic linking is particularly useful for libraries System also known as **shared libraries**

### Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped and Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

System maintains a **ready queue** of ready-to-run processes which have memory images on disk



### Schematic View of Swapping

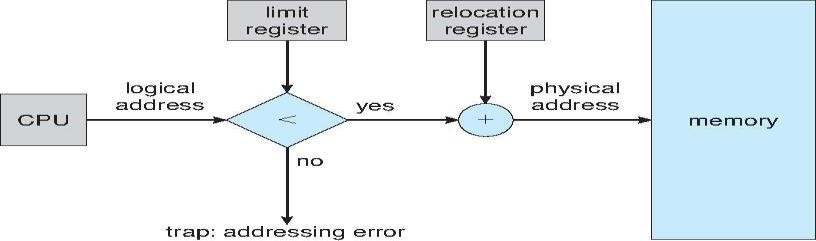
**Contiguous Allocation**

Main memory usually into two partitions:

* Resident operating system, usually held in low memory with interrupt vector
* User processes then held in high memory
* Relocation registers used to protect user processes from each other, and from changing operating-system code and data
* Base register contains value of smallest physical address
* Limit register contains range of logical addresses – each logical address must be less than the limit register MMU maps logical address *dynamically*

### 

### *Hardware Support for Relocation and Limit Registers*



* Multiple-partition allocation
* Hole – block of available memory; holes of various size are scattered throughout memory
* When a process arrives, it is allocated memory from a hole large enough to accommodate it

**Contiguous memory allocation** is one of the efficient ways of allocating main memory to the processes. The memory is divided into two partitions. One for the Operating System and another for the user processes. Operating System is placed in low or high memory depending on the interrupt vector placed. In contiguous memory allocation each process is contained in a single contiguous section of memory.

**Memory protection**

Memory protection is required to protect Operating System from the user processes and user processes from one another. A relocation register contains the value of the smallest physical address for example say 100040. The limit register contains the range of logical address for example say 74600. Each logical address must be less than limit register. If a logical address is greater than the limit register, then there is an addressing error and it is trapped. The limit register hence offers memory protection.

The MMU, that is, Memory Management Unit maps the logical address dynamically, that is at run time, by adding the logical address to the value in relocation register. This added value is the physical memory address which is sent to the memory.

The CPU scheduler selects a process for execution and a dispatcher loads the limit and relocation registers with correct values. The advantage of relocation register is that it provides an efficient way to allow the Operating System size to change dynamically.

**Memory allocation**

There are two methods namely, multiple partition method and a general fixed partition method. In multiple partition method, the memory is divided into several fixed size partitions. One process occupies each partition. This scheme is rarely used nowadays. Degree of multiprogramming depends on the number of partitions. Degree of multiprogramming is the number of programs that are in the main memory. The CPU is never left idle in multiprogramming. This was used by IBM OS/360 called MFT. MFT stands for Multiprogramming with a Fixed number of Tasks.

Generalization of fixed partition scheme is used in MVT. MVT stands for Multiprogramming with a Variable number of Tasks. The Operating System keeps track of which parts of memory are available and which is occupied. This is done with the help of a table that is maintained by the Operating System. Initially the whole of the available memory is treated as one large block of memory called a **hole**.

The programs that enter a system are maintained in an input queue. From the hole, blocks of main memory are allocated to the programs in the input queue. If the hole is large, then it is split into two, and one half is allocated to the arriving process and the other half is returned. As and when memory is allocated, a set of holes in scattered. If holes are adjacent, they can be merged.

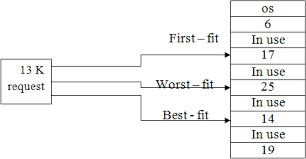
Now there comes a general dynamic storage allocation problem. The following are the solutions to the dynamic storage allocation problem.

**First fit:** The first hole that is large enough is allocated. Searching for the holes starts from the beginning of the set of holes or from where the previous first fit search ended.

**Best fit:** The smallest hole that is big enough to accommodate the incoming process is allocated. If the available holes are ordered, then the searching can be reduced.

**Worst fit:** The largest of the available holes is allocated.

## Example:



First and best fits decrease time and storage utilization. First fit is generally faster.

**Fragmentation**

The disadvantage of contiguous memory allocation is **fragmentation**. There are two types of fragmentation, namely, internal fragmentation and External fragmentation.

**Internal fragmentation**

When memory is free internally, that is inside a process but it cannot be used, we call that fragment as internal fragment. For example say a hole of size 18464 bytes is available. Let the size of the process be 18462. If the hole is allocated to this process, then two bytes are left which is not used. These two bytes which cannot be used forms the internal fragmentation. The worst part of it is that the overhead to maintain these two bytes is more than two bytes.

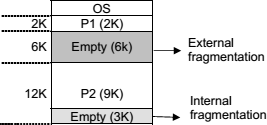
**External fragmentation**

All the three dynamic storage allocation methods discussed above suffer external fragmentation. When the total memory space that is got by adding the scattered holes is sufficient to satisfy a request but it is not available contiguously, then this type of fragmentation is called external fragmentation.

The solution to this kind of external fragmentation is compaction. **Compaction** is a method by which all free memory that are scattered are placed together in one large memory block. It is to be noted that compaction cannot be done if relocation is done at compile time or assembly time. It is possible only if dynamic relocation is done, that is relocation at execution time.

One more solution to external fragmentation is to have the logical address space and physical address space to be non contiguous. Paging and Segmentation are popular non contiguous allocation methods.

## Example for internal and external fragmentation



**Paging**

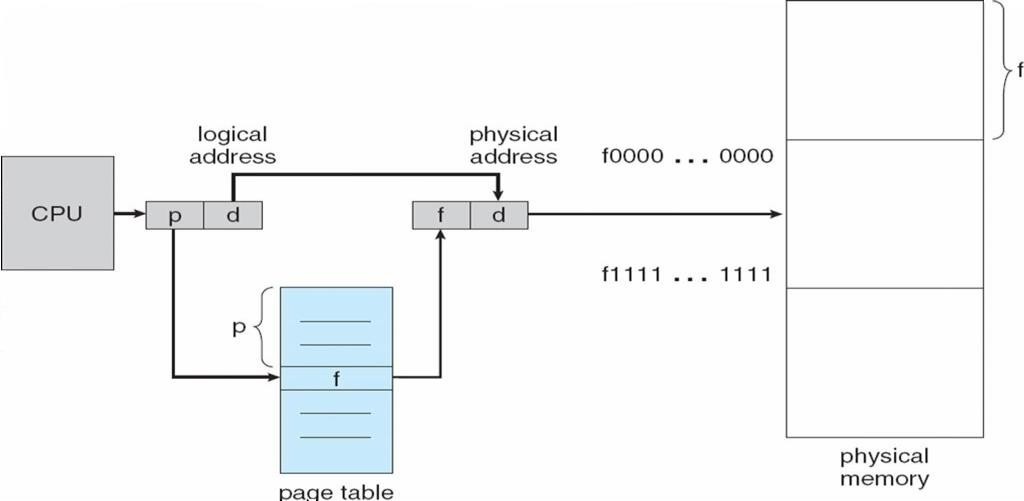
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

### 

### *Paging Hardware*



**Address Translation**

Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

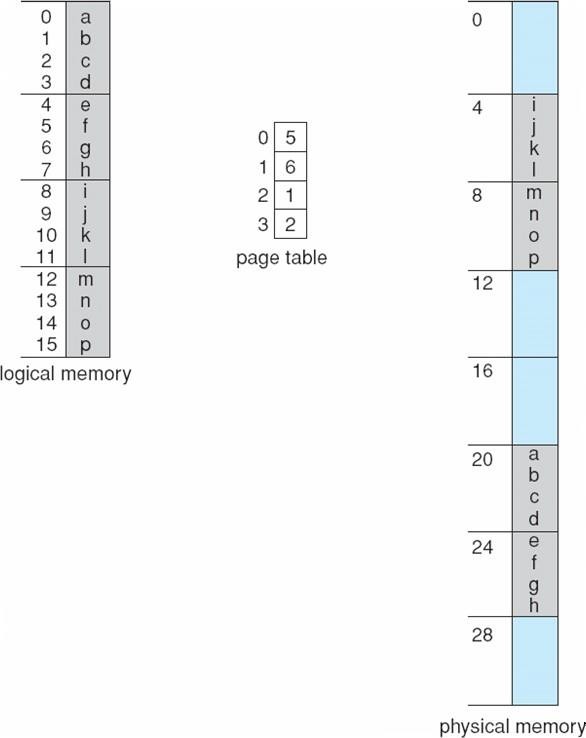
Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.

## Paging Model of Logical and Physical Memory

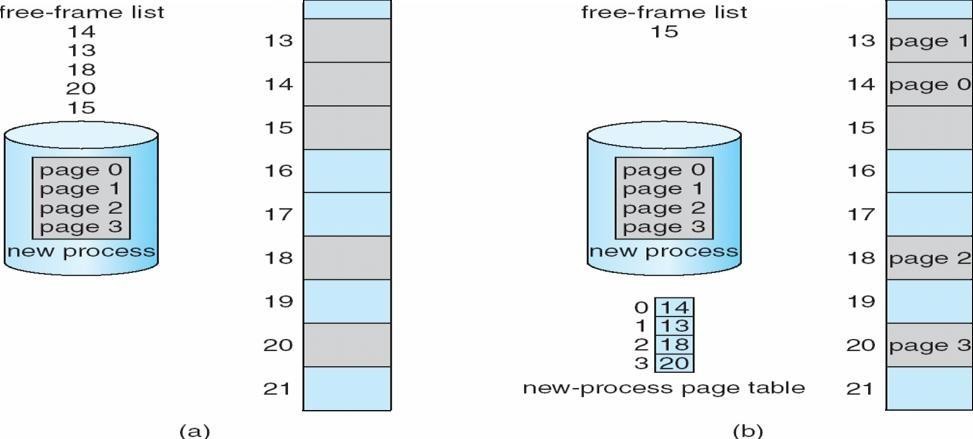
## 

**Paging Example**



**32-byte memory and 4-byte pages**

### *Free Frames*



When the system allocates a frame to any page, it translates this logical address into a physical address and creates entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

## Implementation of Page Table

## Page table is kept in main memory

## Page-table base register (PTBR) points to the page table

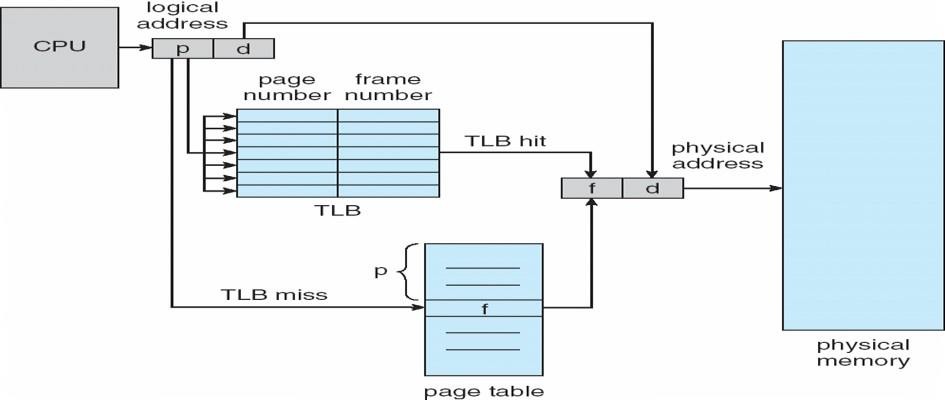
## Page-table length register (PRLR) indicates size of the page table

In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

### 

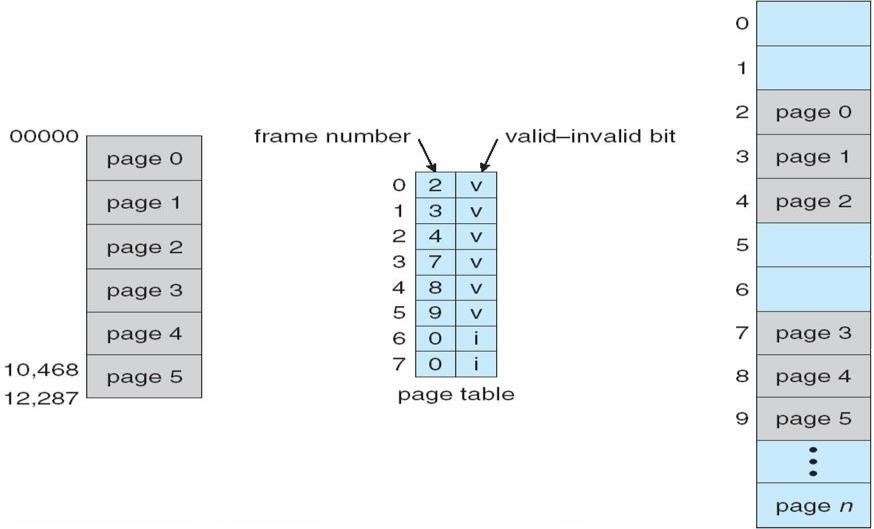
### *Paging Hardware With TLB*



***Memory Protection***

* Memory protection implemented by associating protection bit with each frame
* **Valid-invalid** bit attached to each entry in the page table:
  + “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  + “invalid” indicates that the page is not in the process’ logical address space

**Valid (v) or Invalid (i) Bit In A Page Table**



## Shared Pages Shared code

## One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

## Shared code must appear in same location in the logical address space of all processes.

### *Private code and data*

* Each process keeps a separate copy of the code and data
* The pages for the private code and data can appear anywhere in the logical address space

### 

### *Shared Pages Exampl**e*

**Structure of the Page Table**

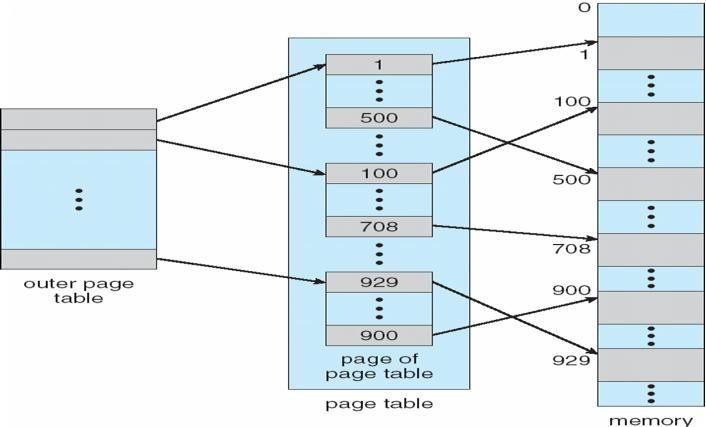
* Hierarchical Paging
* Hashed Page Tables
* Inverted Page Tables

### *Hierarchical Page Tables*

Break up the logical address space into multiple page tables A simple technique is a two-level page table

### 

### *Two-Level Page-Table Scheme*



**Two-Level Paging Example**

A logical address (on 32-bit machine with 1K page size) is divided into:

* a page number consisting of 22 bits
* a page offset consisting of 10 bits

Since the page table is paged, the page number is further divided into:

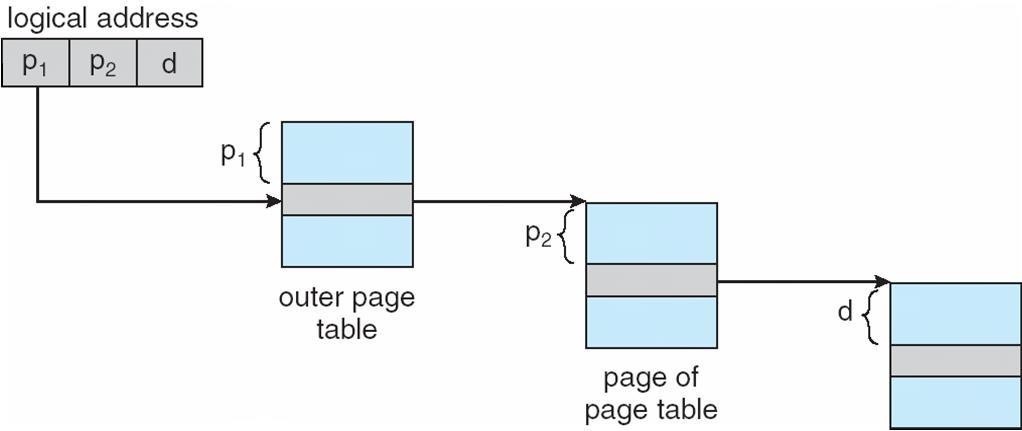
* a 12-bit page number a 10-bit page offset.

Thus, a logical address is as follows:

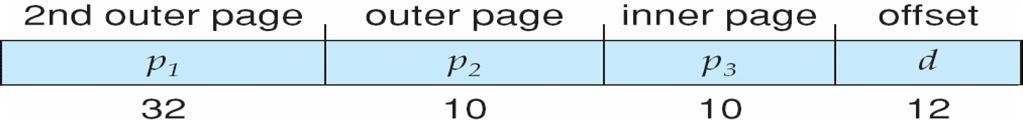
where *pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table

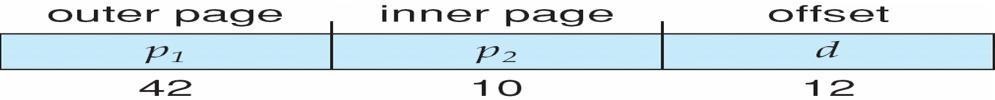
|  |  |  |
| --- | --- | --- |
| Page number | | page offset |
| *p*i  12 | *p*2  10 | *d*10 |

### *Address-Translation Scheme*



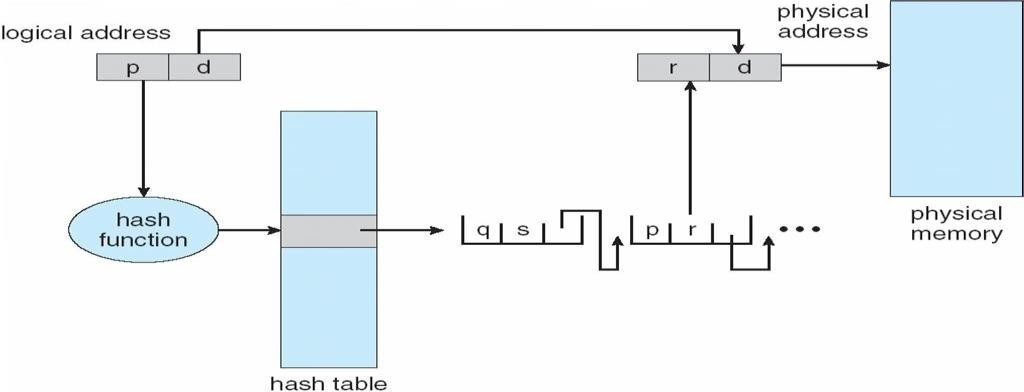
**Three-level Paging Scheme**





**Hashed Page Tables**

* Common in address spaces > 32 bits
* The virtual page number is hashed into a page table
* This page table contains a chain of elements hashing to the same location Virtual page numbers are compared in this chain searching for a match
* If a match is found, the corresponding physical frame is extracted



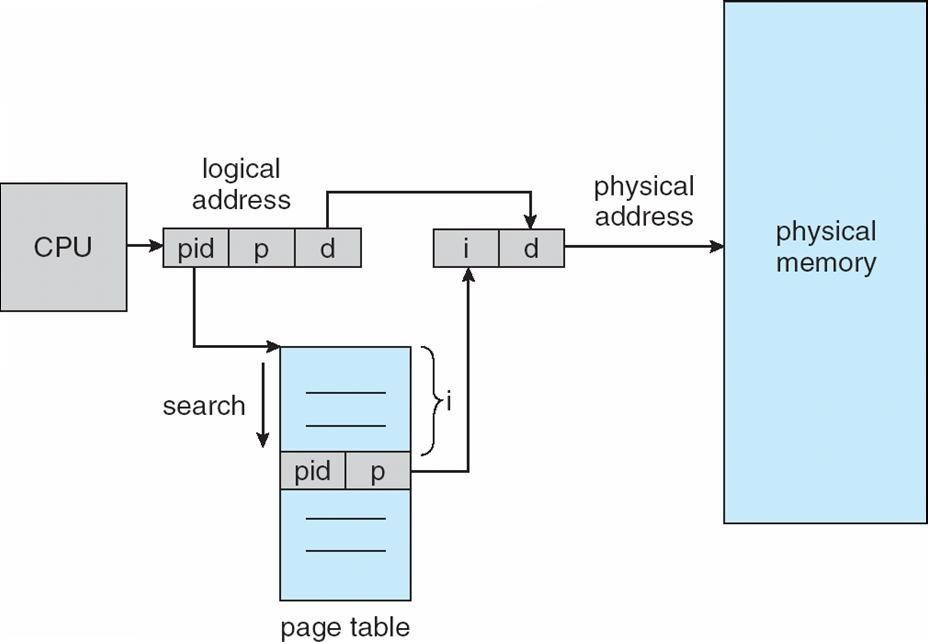
## Hashed Page Table

### *Inverted Page Table*

* One entry for each real page of memory
* Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
* Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
* Use hash table to limit the search to one — or at most a few — page-table entries

### 

### *Inverted Page Table Architecture*



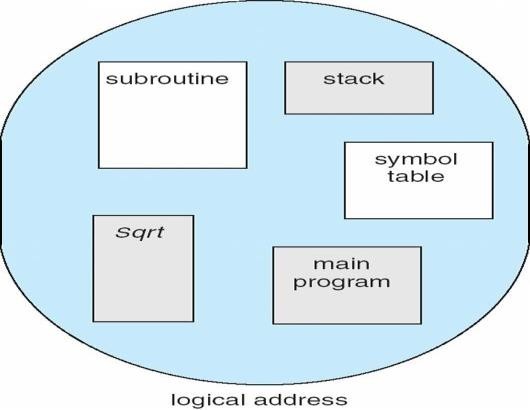
**Advantages and Disadvantages of Paging**

* Paging reduces external fragmentation, but still suffers from internal fragmentation.
* Paging is simple to implement and assumed as an efficient memory management technique.
* Due to equal size of the pages and frames, swapping becomes very easy.
* Page table requires extra memory space, so may not be good for a system having small RAM.

## Segmentation

## Memory-management scheme that supports user view of memory A program is a collection of segments

A segment is a logical unit such as:

* main program
* Procedure
* function method
* object
* local variables, global variables
* common block
* stack
* symbol table
* arrays

### *User’s View of a Program*

**Segmentation Architecture**

Logical address consists of a two tuple: <segment-number, offset>,

**Segment table** – maps two-dimensional physical address

**base** – contains the starting physical address where the segments reside in memory

**limit** – specifies the length of the segment

**Segment-table base register (STBR)** points to the segment table’s location in memory

**Segment-table length register (STLR)** indicates number of segments used by a program;

segment number ***s*** is legal if ***s*** < **STLR**

Protection With each entry in segment table associate:

Validation bit = 0 Þ illegal segment

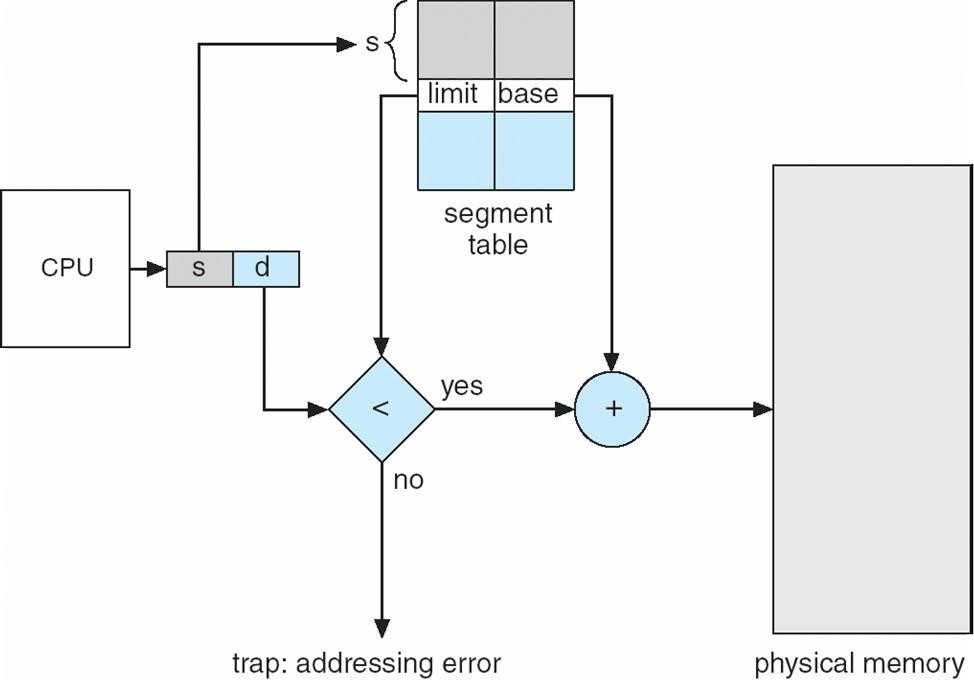
read/write/execute privileges

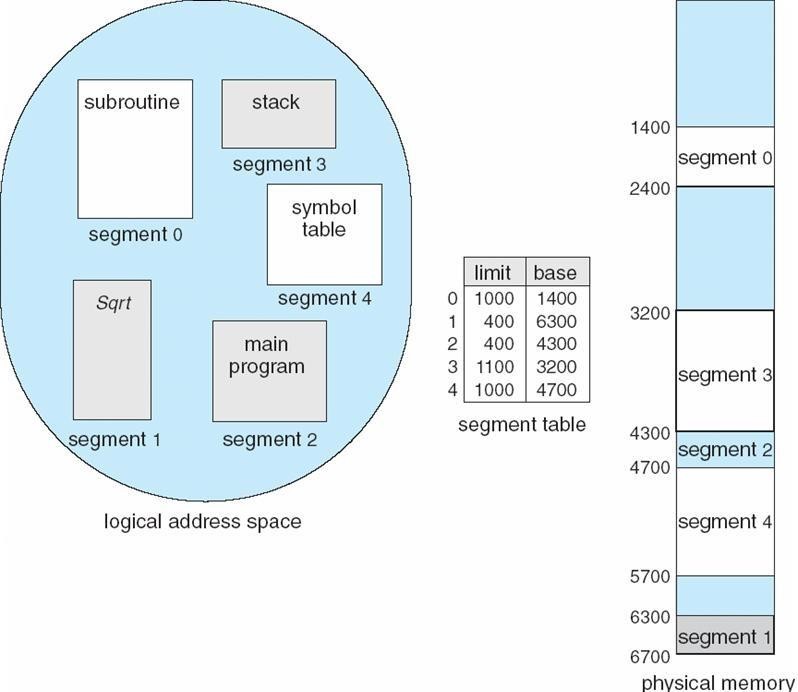
Protection bits associated with segments; code sharing occurs at segment level

Since segments vary in length, memory allocation is a dynamic storage-allocation problem

A segmentation example is shown in the following diagram

### *Segmentation Hardware*



**Example of Segmentation**

**Segmentation with paging**

Instead of an actual memory location the segment information includes the address of a [page](https://en.wikipedia.org/wiki/Page_table) [table](https://en.wikipedia.org/wiki/Page_table) for the segment. When a program references a memory location the offset is translated to a memory address using the page table. A segment can be extended simply by allocating another memory page and adding it to the segment's page table.

An implementation of [virtual memory](https://en.wikipedia.org/wiki/Virtual_memory) on a system using segmentation with paging usually only moves individual pages back and forth between main memory and secondary storage, similar to a paged non-segmented system. Pages of the segment can be located anywhere in main memory and need not be contiguous. This usually results in a reduced amount of input/output between primary and secondary storage and reduced memory fragmentation.