MODULE –II

DEADLOCK & MEMORY MANAGEMENT

Deadlock - Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance,

Deadlock detection, Recovery from deadlock- Main Memory - Swapping - Contiguous Memory

Allocation – Paging - Structure of the Page Table -Segmentation, Segmentation with paging; Virtual

Memory - Demand Paging – Copy on Write- Allocation of Frames – Thrashing

priority over less fortunate ones.

- The system must have priority scheduling, and real-time processes must have the highest priority.
- The priority of real-time processes must not degrade over time, even though the priority of non-real-time processes may.
- Dispatch latency must be small. The smaller the latency, the faster a real- time process can start executing.
- The high-priority process would be waiting for a lower-priority one to finish. This situation is known as **priority inversion**.

DEAD LOCK

Definition:
A process request resources, if the resources are not available at that time, the process enters in to a wait state. It may happen that waiting processes will never again change the state, because the resources they have requested are held by other waiting processes. *This situation is called as **dead lock.***

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.
- CPUcycles, files,and I/O devices (such as printers and DVD drives) are examples of resource types.

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

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**. The process requests the resource. If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use**. The process can operate on the resource
3. **Release**. The process releases the resource.

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

Necessary Conditions for Deadlock:-

A dead lock situation can arise if the following four conditions hold simultaneously in a system.
1) MUTUAL EXCLUSION:- At least one resource must be held in a on-sharable mode. i.e only

one process can hold this resource at a time . other requesting processes should wait till it is released.

2) HOLD & WAIT:- there must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

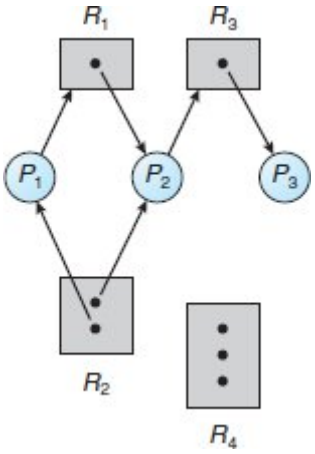
3) NO PREEMPTION:- Resources cannot be preempted, that is a resource can be released voluntarily by the process holding it, after that process has completed its task.

4) CIRCULAR WAIT:- There must exist a set {p0,p1,p1….pn} of waiting processes such that p0 is waiting for a resource that is held by the p1, p1 is waiting for the resource that is held by the p2…. And so on. pn is waiting for a resource that is held by the p0.

Resource-Allocation Graph

A deadlock can be described in terms of a directed graph called system resource-allocation graph.

- A set of vertices V and a set of edges E.
  - V is partitioned into two types:
    - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
    - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
      - request edge – directed edge Pi → Rj
      - assignment edge – directed edge Rj → P



The resource-allocation graph depicts the following situation.

The sets P, R, and E:

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

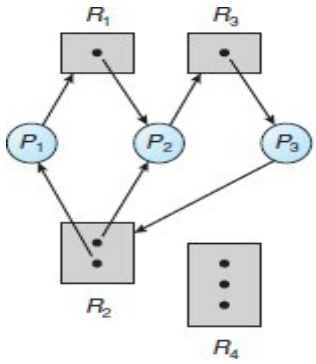
Resource instances:

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

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 an instance of R3.
- Process P3 is holding an instance of R3.

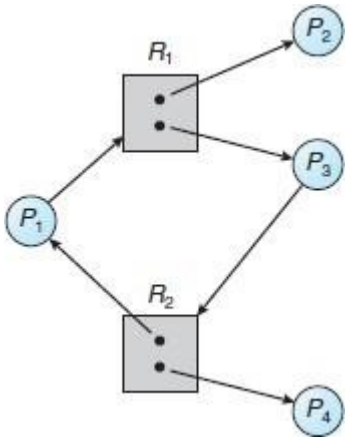
Resource-allocation graph with a deadlock.



- Processes P1, P2, and 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 process 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
- 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.

Resource-allocation graph with a cycle but no deadlock.



Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows.

- **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

DEADLOCK PREVENTION

 For a deadlock to occur, each of the four necessary conditions must hold.
 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.
– For example, a printer cannot be simultaneously shared by several processes.
– A process never needs to wait for a sharable resource.

2. Hold and Wait

– must guarantee that whenever a process requests a resource, it does not hold any other resources.

– One protocol requires each process to request and be allocated all its resources before it begins execution,

– Or another protocol allows a process to request resources only when the process has none. So, before it can request any additional resources, it must release all the resources that it is currently allocated.

3. Denying 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.

4. Denying Circular wait

- Impose a total ordering of all resource types and allow each process to request for resources in an increasing order of enumeration.
- Let R = {R1,R2,...Rm} be the set of resource types.
- Assign to each resource type a unique integer number.
- If the set of resource types R includes tapedrives, disk drives and printers.

  F(tapedrive)=1,
  F(diskdrive)=5,
  F(Printer)=12.
- Each process can request resources only in an increasing order of enumeration.

DEADLOCK AVOIDANCE

 An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

 Each request requires that in making this decision the system consider

- the resources currently available,
- the resources currently allocated to each process,
- the future requests and releases of each process.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.

The resource-allocation state is defined by the number of available and allocated 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.

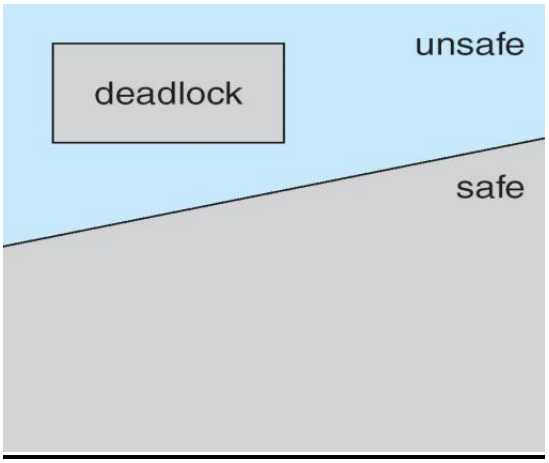
• System is in safe state if there exists a sequence <P1, P2, …, Pn> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j < i.

• 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.



Banker's Algorithm

 The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

 The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

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.
    1. **Available:** indicates the number of available resources of each type.
    2. **Max:** Max[i, j]=k then process Pi may request at most k instances of resource type Rj
    3. **Allocation :** Allocation[i. j]=k, then process Pi is currently allocated K instances of resource type Rj
    4. **Need :** if Need[i, j]=k then process Pi may need K more instances of resource type Rj ,Need [i, j]=Max[i, j]-Allocation[i, j]

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

Safety algorithm

1. Initialize work := available and Finish [i]:=false for i=1,2,3 .. n

2.Find an i such that both

a. Finish[i]=false

b. Needi<= Work if no such i exists, goto step 4

3. work :=work+ allocationi; Finish[i]:=true goto step 2

4. If finish[i]=true for all i, then the system is in a safe state

Example:

Given the following state for the Banker's Algorithm.

5 processes P0 through P4

3 resource types A (6 instances), B (9 instances) and C (5 instances).

Snapshot at time T0:

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

a) Calculate the available vector.

b) Calculate the Need matrix.

c) Is the system in a safe state? If so, show one sequence of processes which allows the system to complete. If not, explain why.

d) Given the request (1, 2, 0) from Process P2. Should this request be granted? Why or why not?

a) Calculate the available vector.

Available
A B C
1 2 0

b) Calculate the Need matrix.

	Need
	A B C
P_0	5 6 2
P_1	1 1 0
P_2	2 3 3
P_3	0 1 1
P_4	3 5 2

c) Is the system in a safe state? If so, show one sequence of processes which allows the system to complete. If not, explain why.

1. Initialize the *Work* and *Finish* vectors.

Work = *Available* = (1, 2, 0)

Finish = (false, false, false, false, false)

2. Find index *i* such that *Finish*[*i*] = false and *Needi* <= *Work*

i	*Work* = *Work* + *Allocation*$_i$	*Finish*
1	(1, 2, 0) + (1, 1, 2) = (2, 3, 2)	(false, true, false, false, false)
3	(2, 3, 2) + (2, 1, 1) = (4, 4, 3)	(false, true, false, true, false)
2	(4, 4, 3) + (0, 3, 0) = (4, 7, 3)	(false, true, true, true, false)
4	(4, 7, 3) + (1, 1, 1) = (5, 8, 4)	(false, true, true, true, true)
0	(5, 8, 4) + (1, 1, 1) = (6, 9, 5)	(true, true, true, true, true)

3. Since *Finish*[*i*] = true for all *i*, hence the system is in a safe state. The sequence of processes which allows the system to complete is P1, P3, P2, P4, P0**.**

d) Given the request (1, 2, 0) from Process P2. Should this request be granted? Why or why not?

1. Check that *Request2* <= *Need2*.

Since (1, 2, 0) <= (2, 3, 3), hence, this condition is satisfied.

2. Check that *Request2* <= *Available*.

Since (1, 2, 0) <= (1, 2, 0), hence, this condition is satisfied.

3. Modify the system's state as follows:

Available = *Available* – *Request2* = (1, 2, 0) – (1, 2, 0) = (0, 0, 0)

Allocation2 = *Allocation2* + *Request2* = (0, 3, 0) + (1, 2, 0) = (1, 5, 0)

Need2 = *Need2* – *Request2* = (2, 3, 3) – (1, 2, 0) = (1, 1, 3)

4. Apply the safety algorithm to check if granting this request leaves the system in a safe state.

1. Initialize the *Work* and *Finish* vectors.

Work = *Available* = (0, 0, 0)

Finish = (false, false, false, false, false)

2. At this point, there does not exist an index *i* such that *Finish*[*i*] = false and *Needi* <= *Work*.

Since *Finish*[*i*] ≠ true for all *i*, hence the system is not in a safe state.

Therefore, this request from process *P2* should not be granted.

Resource-Request Algorithm

Let Requesti be the request vector for process Pi . If Requesti [j] == k, then process Pi wants k instances of resource type Rj . When a request for resources is made by process Pi , the following actions are taken:

1. If Requesti ≤Needi , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If Requesti ≤ Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available–Requesti ;

Allocationi = Allocationi + Requesti ;

Needi = Needi –Requesti ;

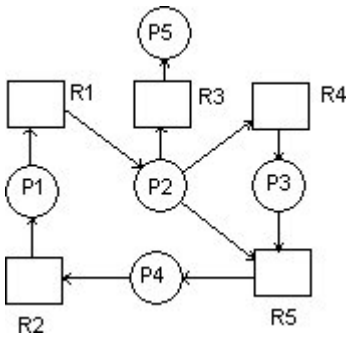
DEADLOCK DETECTION

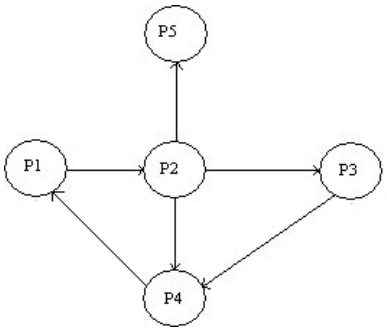
Deadlock Detection

(i) Single instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algorithm that use a variant of resource-allocation graph called a wait for graph.

Resource Allocation Graph **Wait for Graph**





(ii) Several Instance of a resource type

Available : Number of available resources of each type

Allocation : number of resources of each type currently allocated to each process

Request : Current request of each process

If Request [i,j]=k, then process P_i is requesting K more instances of resource typeR_j.

1. Initialize work := available
   Finish[i]=false, otherwise finish [i]:=true
2. Find an index i such that both
   a. Finish[i]=false
   b. $Request_i$<=work
   if no such i exists go to step4.
3. Work:=work+$allocation_i$
   Finish[i]:=true goto step2
4. If finish[i]=false then process P_i is deadlocked

DEADLOCK RECOVERY.

- There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

1. Process Termination

Two basic approaches, both of which recover resources allocated to terminated processes:

Ⓟ Terminate all processes involved in the deadlock. This definitely solves the

deadlock, but at the expense of terminating more processes than would be absolutely necessary.

ⓅTerminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

ⓅIn the latter case there are many factors that can go into deciding which processes to terminate next:

- Process priorities.
- How long the process has been running, and how close it is to finishing.
- How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
    1. How many more resources does the process need to complete.
    2. How many processes will need to be terminated
    3. Whether the process is interactive or batch.
    4. (Whether or not the process has made non-restorable changes to any resource.)

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 - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

WINDOWS 7 – THREAD AND SMP MANAGEMENT

The native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality.

Characteristics of Windows processes:

- Windows processes are implemented as objects.
- A process can be created as new process, or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

A Windows Process and Its Resources

- Each process is assigned a security access token, called the primary token of the process. When a user first logs on, Windows creates an access token that includes the security ID for the user.

.

1. MEMORY MANAGEMENT: BACKGROUND

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes.

It decides which process will get memory at what time.

It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

1.1 Basic Hardware

Program must be brought (from disk) into memory and placed within a process for it to be run

• Main memory and registers are only storage CPU can access directly

• Memory unit only sees a stream of addresses + read requests, or address + data and write requests

• Register access in one CPU clock (or less)

• Main memory can take many cycles, causing a **stall**

• Cache sits between main memory and CPU registers

• Protection of memory required to ensure correct operation

Ⓟ

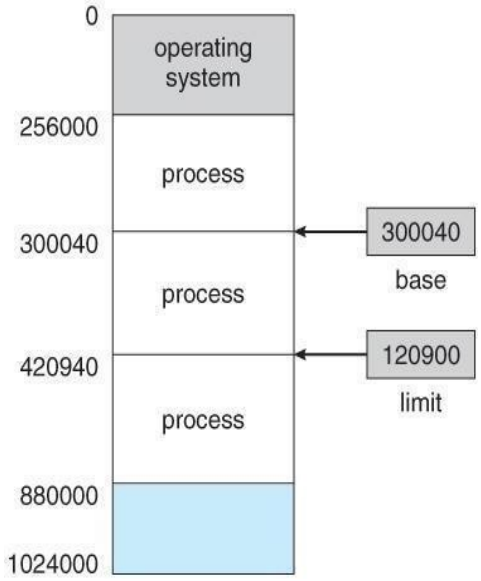
We can provide this protection by using two registers, usually a **base** and a **limit**

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 300040 and limit register is 120900, then the

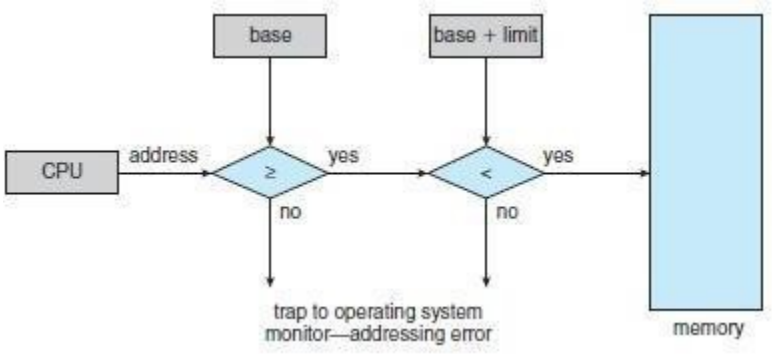
program can legally access all addresses from 300040 through 420940 (inclusive).



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system memory or other users' 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.



1.2 Address Binding

Definition

Converting the address used in a program to an actual physical address.

Address binding is the process of mapping the program's logical or virtual addresses to corresponding physical or main memory addresses.

In other words, a given logical address is mapped by the MMU (Memory Management Unit) to a physical address.

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:

Three different stages of binding:

1. **Compile time**. The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute code can be generated (Static).

2. **Load time**. The compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code (Static).

3. **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. The absolute addresses are generated by hardware. Most general-purpose OS use this method (Dynamic).

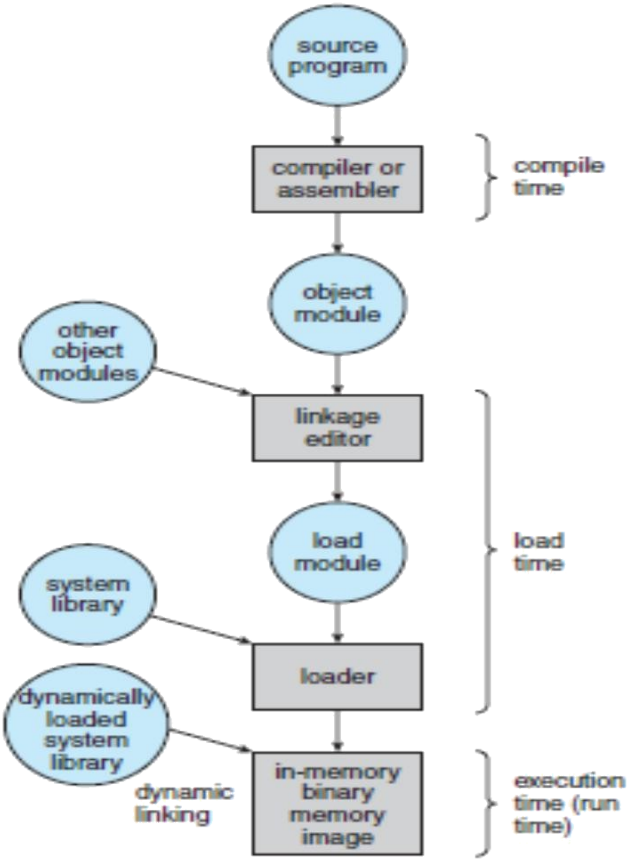


Figure 8.3 Multistep processing of a user program.

1.3 Logical vs. Physical Address Space

Logical address – generated by the CPU; also referred to as **"virtual address" Physical address** – address seen by the memory unit.

Logical and physical addresses are the **same** in compile-time and load-time address-binding schemes

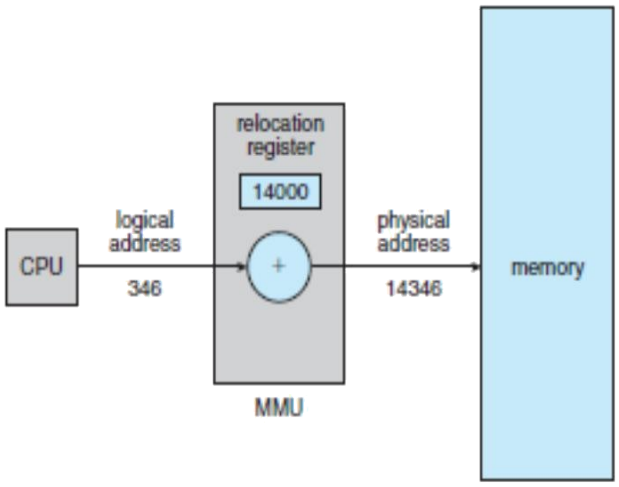
Logical (virtual) and physical addresses **differ** in execution-time address- binding scheme

Memory-Management Unit (MMU)

It is a hardware device that maps virtual / Logical address to physical address.
In this scheme, the relocation register's value is added to Logical address generated by a user process.

The Base register is called a relocation register.

- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses -all as the number 346.
- The user program deals with logical addresses.



1.4 Dynamic Loading

Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory, retrieve the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory.

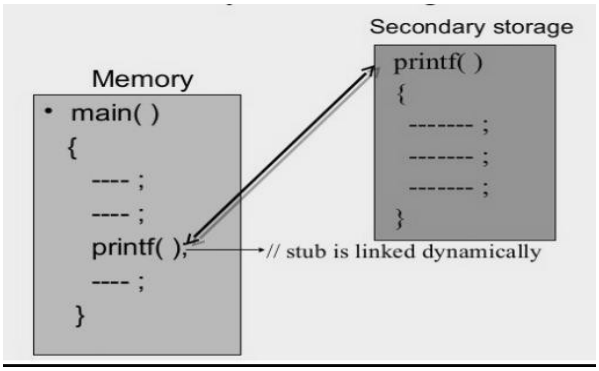
Dynamic loading means loading the library (or any other binary for that matter) into the memory during load or run-time.

Dynamic loading can be imagined to be similar to plugins, that is an exe can actually execute before the dynamic loading happens(The dynamic loading for example can be created using Load Library call in C or C++)

1.5 Dynamic Linking and shared libraries

Dynamic linking refers to the linking that is done during load or run-time and not when the exe is created.
In case of dynamic linking the linker while creating the exe does minimal work. For the dynamic linker to work it actually has to load the libraries too. Hence it's also called linking loader.
Small piece of code, *stub*, used to indicate how to load 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.

- Shared libraries: Programs linked before the new library was installed will continue using the older library.



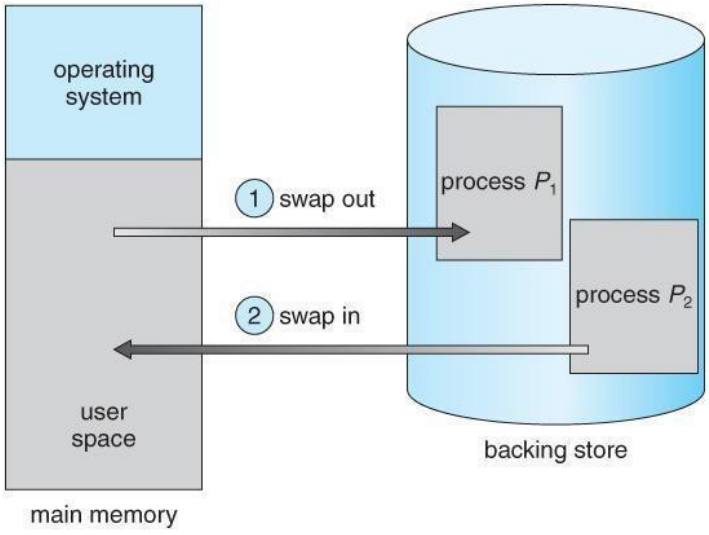
2. SWAPPING

2.1 Basic

- A process can be swapped temporarily out of memory to a backing store (SWAP OUT)and then brought back into memory for continued execution (SWAP IN).
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users & it 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
- **Transfer time:** Major part of swap time is transfer time. Total transfer time is directly proportional to the amount of memory swapped.

□ **Example**: Let us assume the user process is of size 1MB & the backing store is a standard hard disk with a transfer rate of 5MBPS.

Transfer time = 1000KB/5000KB per second
= 1/5 sec = 200ms



A process with dynamic memory requirements will need to issue system calls (request memory() and release memory()) to inform the operating system o f its changing memory needs.

2.2 Swapping on Mobile Systems

Swapping is typically not supported on mobile platforms, for several reasons:

Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.

Flash memory can only be written to a limited number of times before it becomes unreliable.

The bandwidth to flash memory is also lower.

Apple's IOS asks applications to voluntarily free up memory

Read-only data, e.g. code, is simply removed, and reloaded later if needed.

Modified data, e.g. the stack, is never removed.

Apps that fail to free up sufficient memory can be removed by the

OS Android follows a similar strategy.

Prior to terminating a process, Android writes its application state to flash memory for quick restarting.

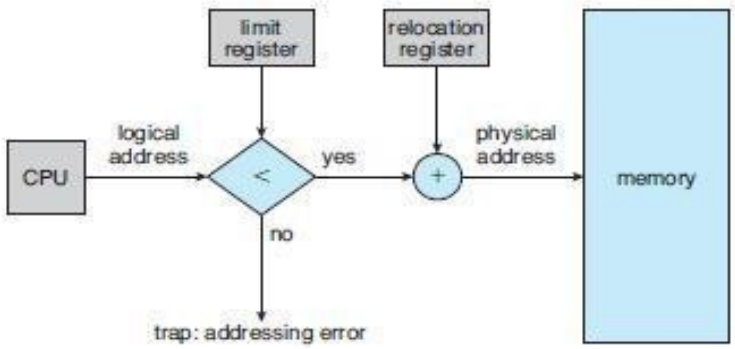
3. CONTIGUOUS MEMORY ALLOCATION

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.

3.1 Memory Protection

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.



3.2 Memory Allocation

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process.

When a partition is free, a process is selected from the input queue and loaded into it.

There are two methods namely:

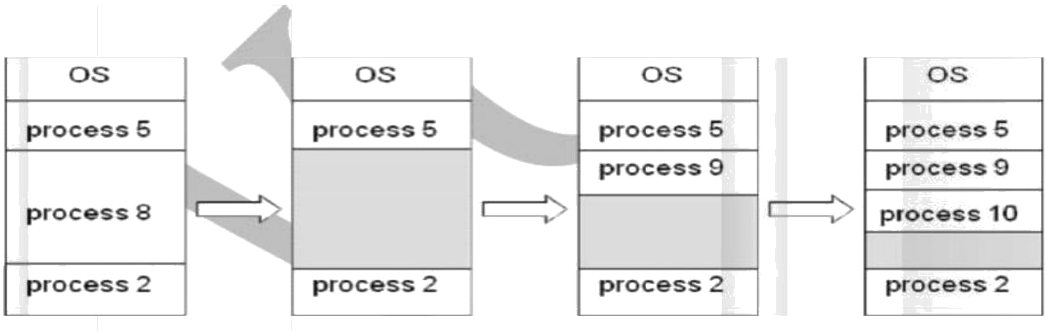
- Fixed – P**a**rtition Method
- Variable – Partition Method

- **Fixed – Partition Method:**

Divide memory into fixed size partitions, where each partition has exactly one process. The drawback is Memory space unused within a partition is wasted.(eg. When process size < partition size)

- **Variable-partition method:**

  - Divide memory into variable size partitions, depending upon the size of the incoming process.
  - When a process terminates, the partition becomes available for another process.
  - As processes complete and leave they create holes in the main memory.
  - ***Hole*** – block of available memory; holes of various size are scattered throughout memory.



Dynamic Storage- Allocation Problem:

How to satisfy a request of size n‘from a list of free holes?

Ⓟ The free blocks of memory are known as holes. The set of holes is searched to determine which hole is best to allocate.

Solution:

o First-fit: Allocate the first hole that is big enough.
o Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
o Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

Example :

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?

a. First-fit:
1. 212K is put in 500K partition
2. 417K is put in 600K partition
3. 112K is put in 288K partition (new partition 288K = 500K − 212K)
4. 426K must wait

b. Best-fit:
1. 212K is put in 300K partition
2. 417K is put in 500K partition
3. 112K is put in 200K partition
4. 426K is put in 600K partition

c. Worst-fit:
1. 212K is put in 600K partition
2. 417K is put in 500K partition
3. 112K is put in 388K partition
4. 426K must wait

In this example, best-fit turns out to be the best.

NOTE: First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

3.3Fragmentation:

Fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both.

1. **External Fragmentation** – This takes place when enough total memory space exists to satisfy a request, but it is not contiguous i.e, storage is fragmented into a large number of small holes scattered throughout the main memory.
2. **Internal Fragmentation** – Allocated memory may be slightly larger than requested memory.

Example: hole = 184 bytes Process size = 182 bytes.

We are left with a hole of 2 bytes.

Ⓟ **Solutions**

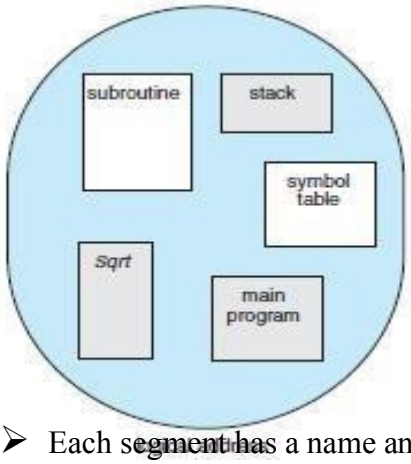
Compaction: Move all processes towards one end of memory, hole towards other end of memory, producing one large hole of available memory. This scheme is expensive as it can be done if relocation is dynamic and done at execution time.

4. SEGMENTATION

4.1 Basic Method

o Memory-management scheme that supports user view of memory

o 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



- Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities:

a segment name and an offset.

A logical address consists of a two tuple:

<segment-number, offset>.

4.2Segmentation Hardware

Each entry in the segment table has a segment base and a segment limit.

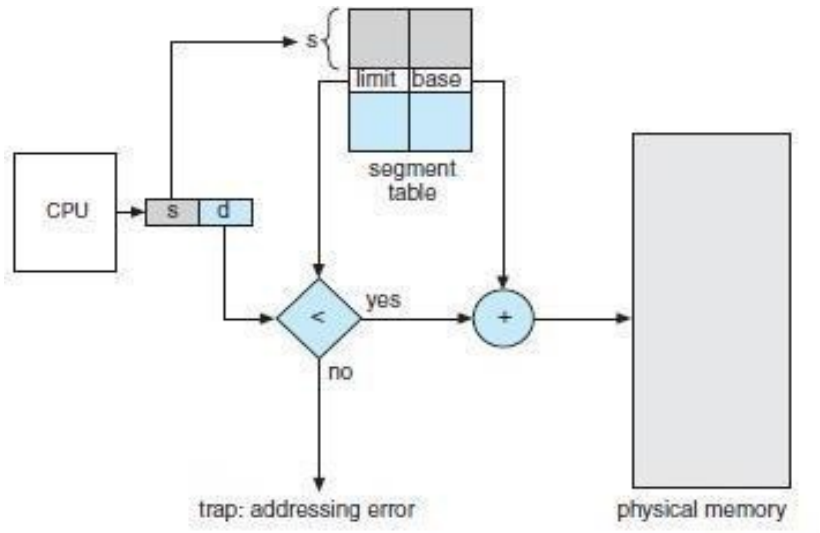
The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment

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.

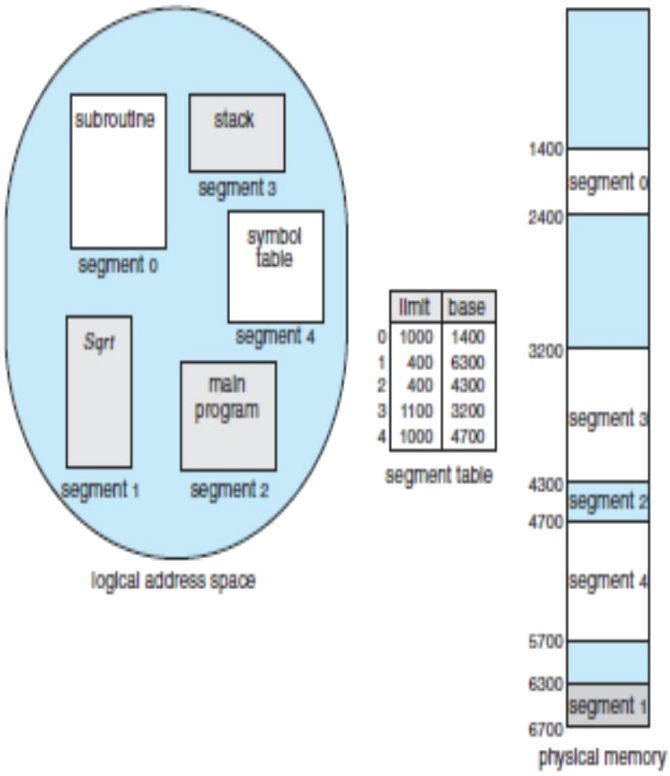
If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.



For example,

segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.



5. PAGING

- It is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

5.1 Basic Method

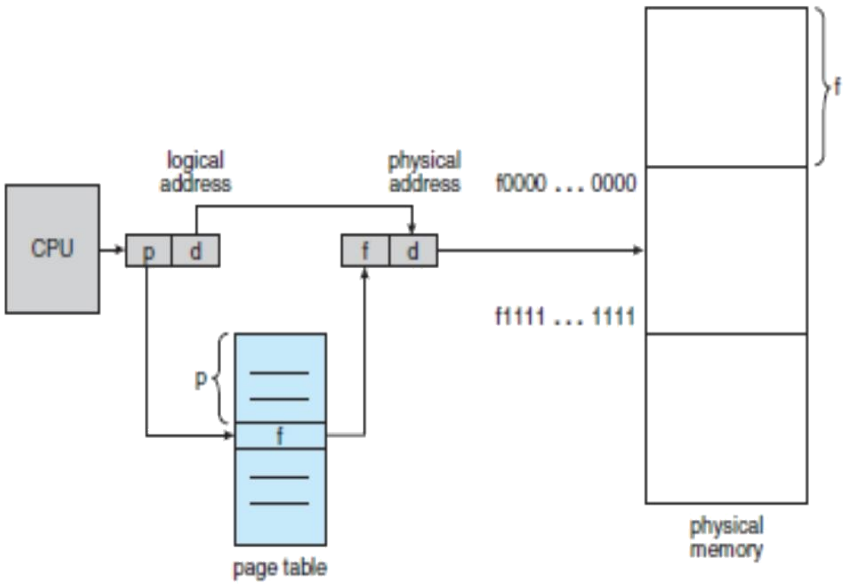
- Divide logical memory into blocks of same size called **"pages"**.
- Divide physical memory into fixed-sized blocks called **"frames"**
- Page size is a power of 2, between 512 bytes and 16MB.

Address Translation Scheme

each page Address generated by CPU (logical address) is divided into:

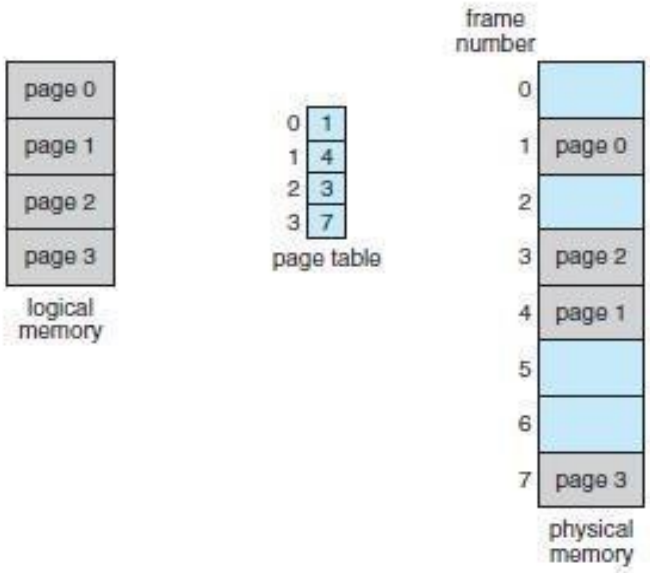
Page number *(p)* – used as an index into a page table which contains base address of in physical memory

Page offset *(d)* – combined with base address to define the physical address
i.e., Physical address = base address + offset



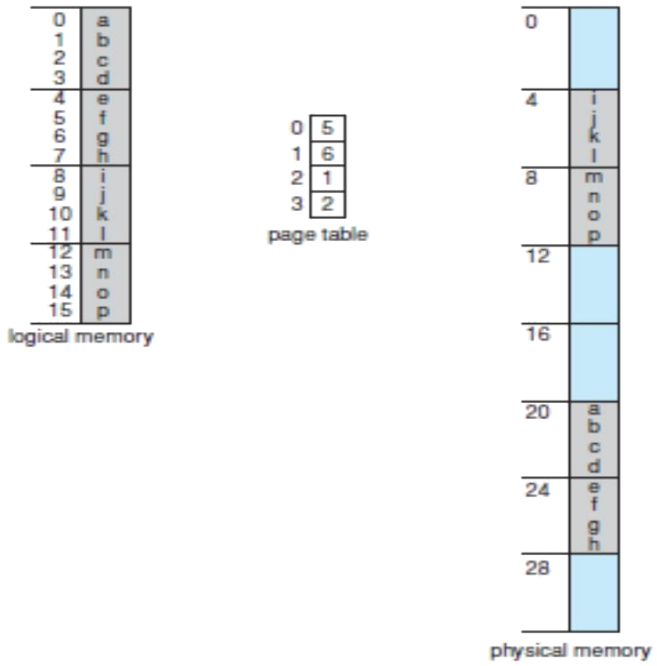
The page number is used as an index into a page table. The page table contains the base address of each page in physical memory.

This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



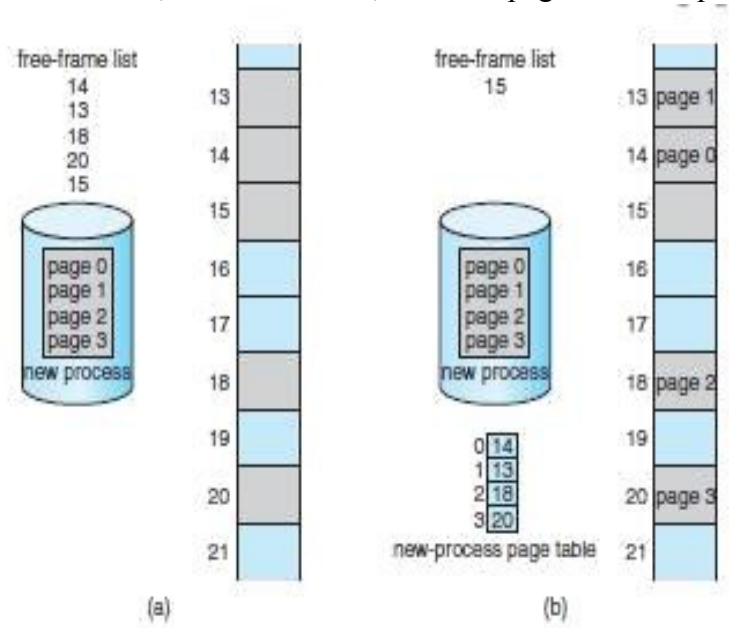
Consider the memory in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.

Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.



Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory, which frames are allocated, which frames are available, how many total frames there are, and so on.

This information is generally kept in a data structure called a frame table.
The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

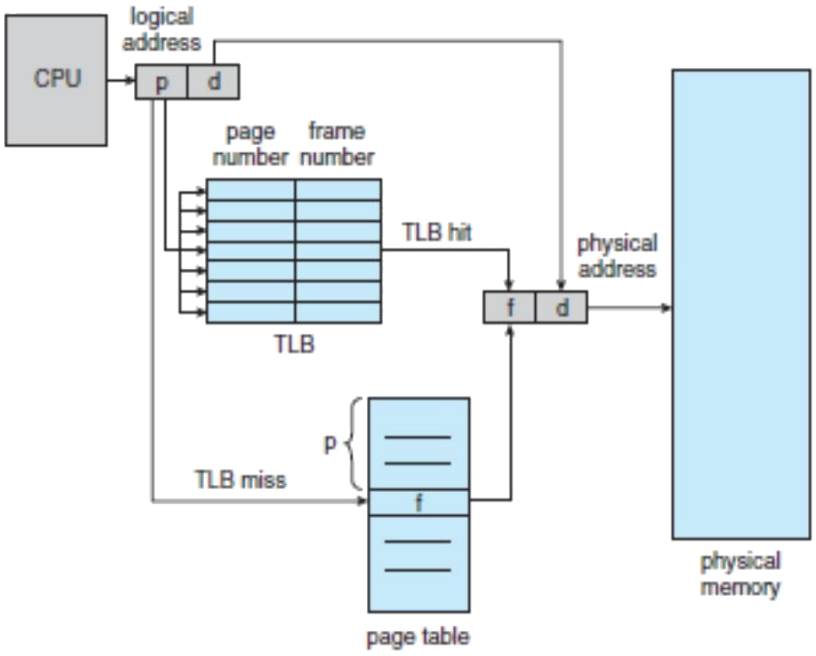


5.2 Hardware Support

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts:
  - a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- The TLB contains only a few of the page-table entries.
  When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.
- Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.
- If the page number is found, its frame number is immediately available and is used to access Memory.
  Hit Ratio - The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.

- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds.
  effective access time = $0.80 \times 100 + 0.20 \times 200$
  = 120 nanoseconds
  For a 99-percent hit ratio, which is much more realistic, we have effective access time = $0.99 \times 100 + 0.01 \times 200 = 101$ nanoseconds

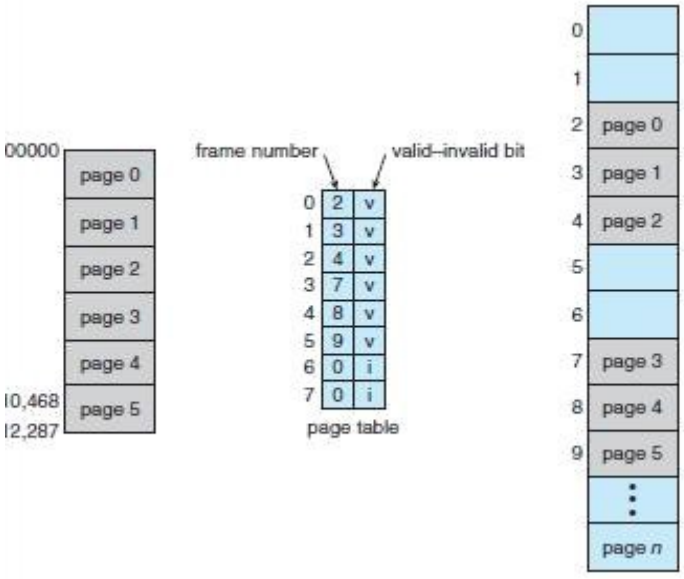


5.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame.
One additional bit is generally attached to each entry in the page table: a valid–invalid bit.

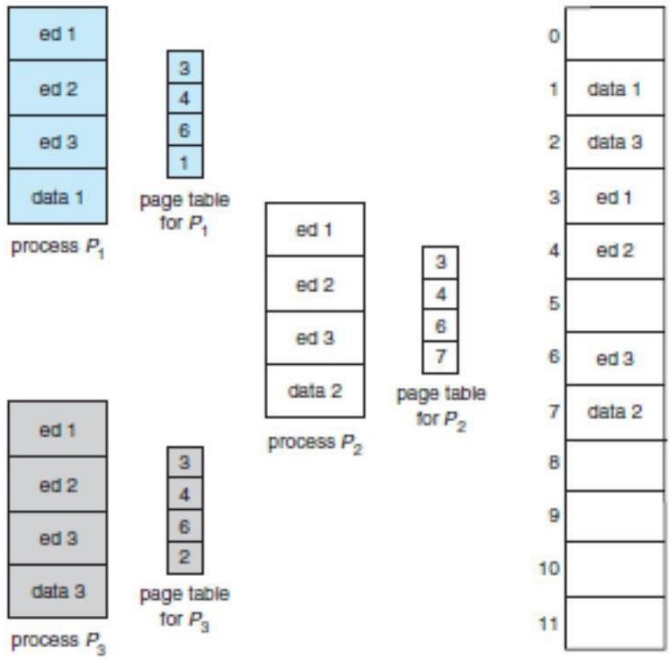
When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page.

When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit.



5.4 Shared Pages

An advantage of paging is the possibility of sharing common code.



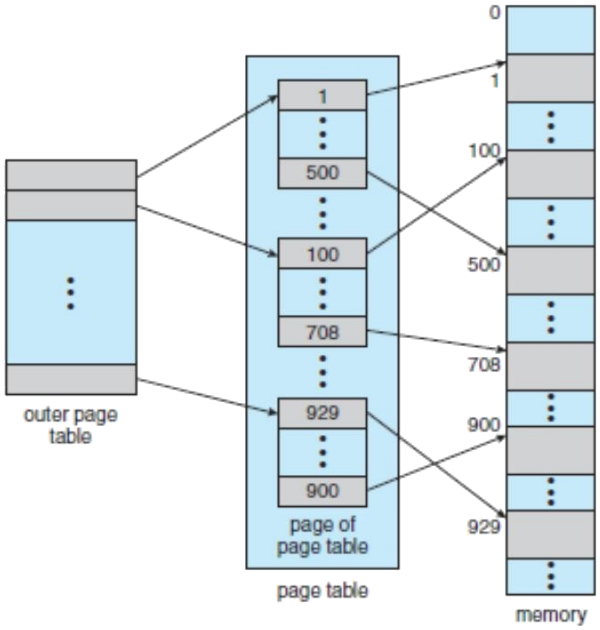
6. STRUCTURE OF PAGE TABLE

The most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

1. Hierarchical Paging

The page table itself becomes large for computers with large logical address space (232 to 264).
Example:

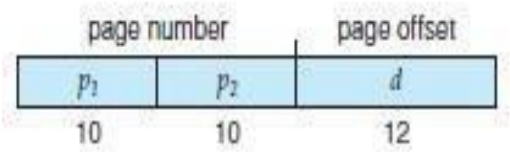
- Consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB(212), then a page table may consist of up to 1 million entries (232/212).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- The page table should be allocated contiguously in main memory.
- The solution to this problem is to divide the page table into smaller pieces.
- One way of dividing the page table is to use a two-level paging algorithm,



For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12
bits.

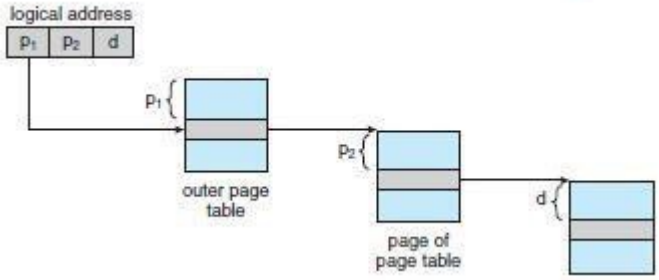
Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

Thus, a logical address is as follows:



where
p1 - an index into the outer page table
p2 - the displacement within the page of the inner page table.

The address-translation method for this architecture is shown in the figure. Because address translation
works from the outer page table inward, this scheme is also known as a forward-mapped page table.



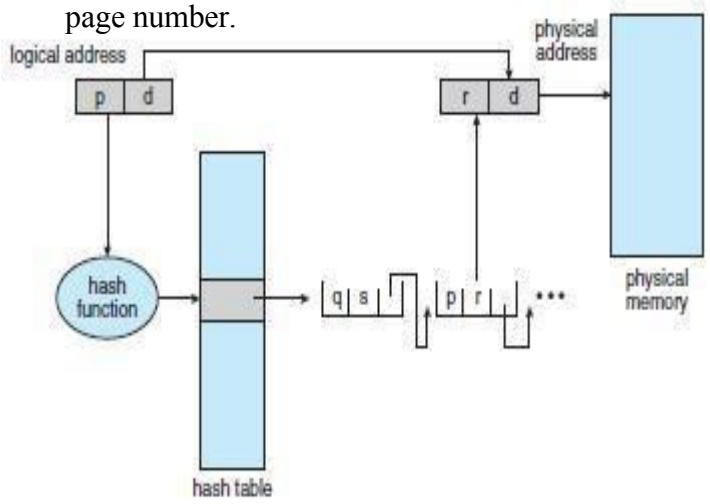
2. Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).

- Each element consists of three fields:
  - The virtual page number
  - The value of the mapped page frame
  - A pointer to the next element in the linked list.

Algorithm:

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



3. Inverted Page Table

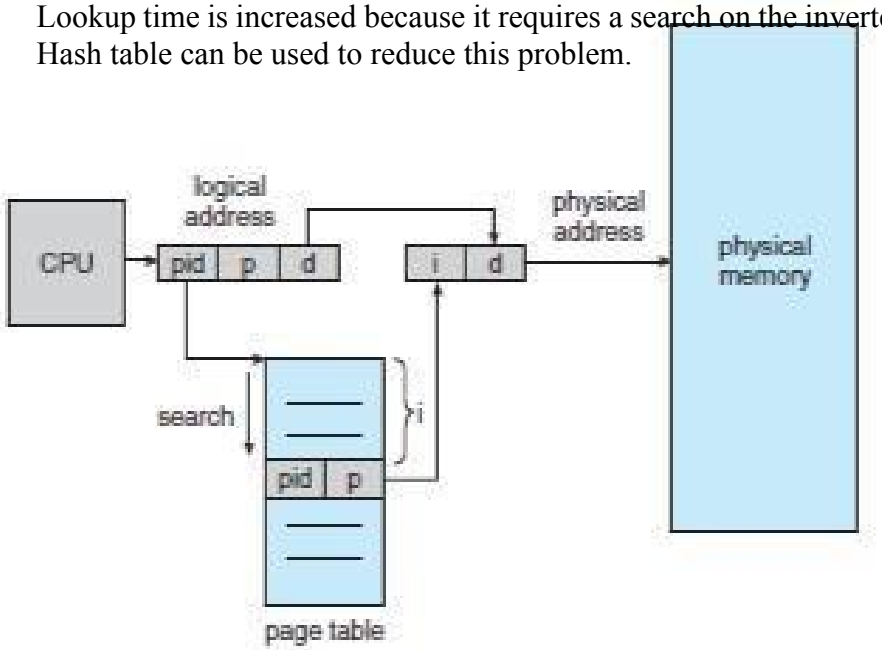
With each process having its own page table, and with each page table consuming considerable amount of memory

We use a lot of memory to keep track of memory.

Inverted page table has one entry for each real page of memory.

Lookup time is increased because it requires a search on the inverted table.

Hash table can be used to reduce this problem.

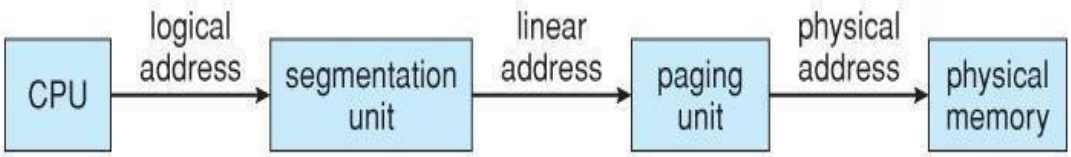


Each virtual address in the system consists of a triple:
<process-id, page-number, offset>.

7.INTEL 32 AND 64-BIT ARCHITECTURES

IA-32 Segmentation

The Pentium CPU provides both pure segmentation and segmentation with paging. In the latter case, the CPU generates a logical address (segment-offset pair), which the segmentation unit converts into a logical linear address, which in turn is mapped to a physical frame by the paging unit



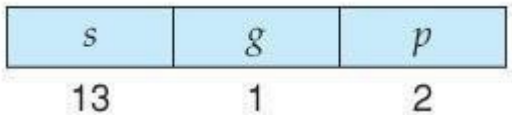
IA-32 Segmentation

The Pentium architecture allows segments to be as large as 4 GB, (24 bits of offset).
Processes can have as many as 16K segments, divided into two 8K groups:

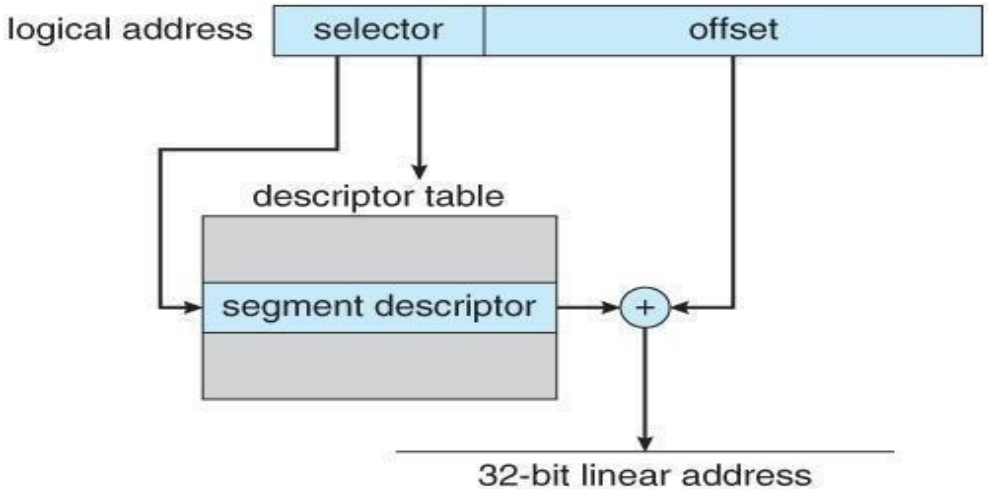
- 8K private to that particular process, stored in the Local Descriptor Table, LDT.
- 8K shared among all processes, stored in the Global Descriptor Table, GDT.

Logical addresses are (selector, offset) pairs, where the selector is made up of 16 bits:

- A 13 bit segment number (up to 8K)
- A 1 bit flag for LDT vs. GDT.
- 2 bits for protection codes.

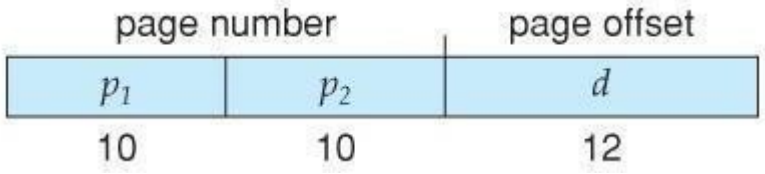


The descriptor tables contain 8-byte descriptions of each segment, including base and limit registers. Logical linear addresses are generated by looking the selector up in the descriptor table and adding the appropriate base address to the offset.



IA-32 Paging

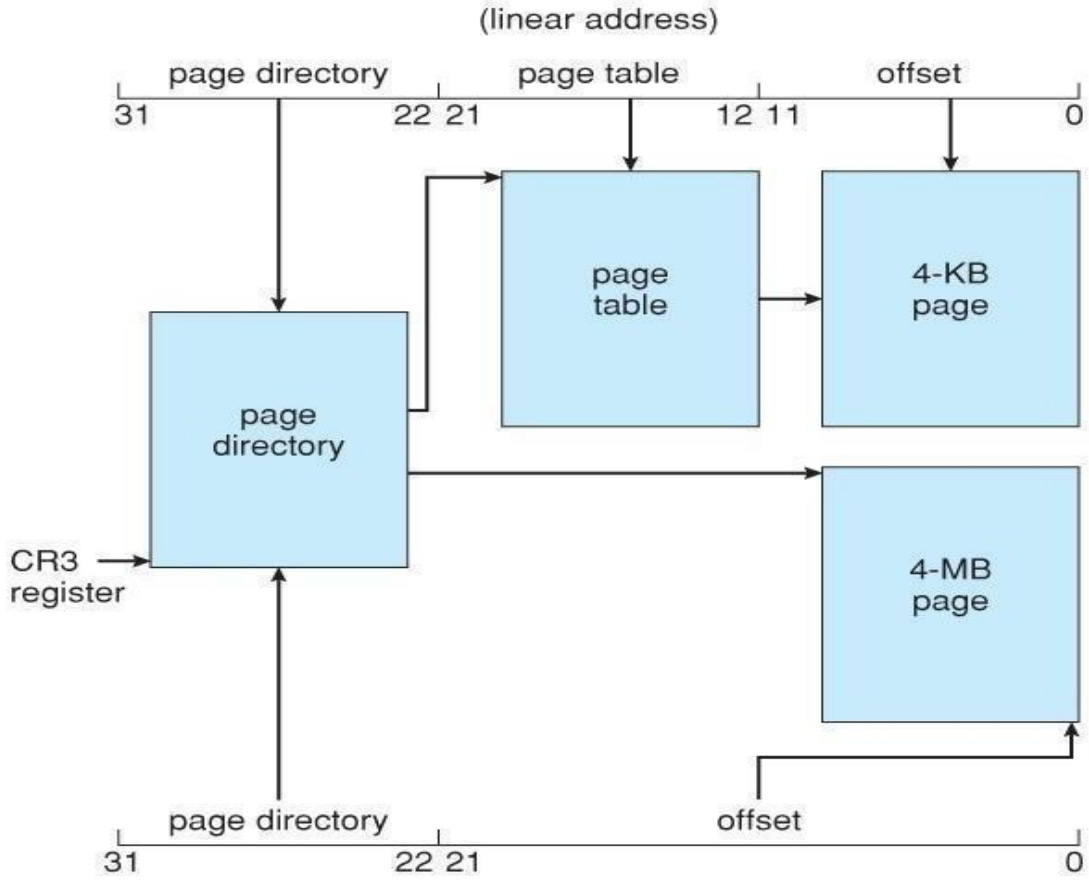
Pentium paging normally uses a two-tier paging scheme, with the first 10 bits being a page number for an outer page table (a.k.a. page directory), and the next 10 bits being a page number within one of the 1024 inner page tables, leaving the remaining 12 bits as an offset into a 4K page.



A special bit in the page directory can indicate that this page is a 4MB page, in which case the remaining 22 bits are all used as offset and the inner tier of page tables is not used.

The CR3 register points to the page directory for the current process.

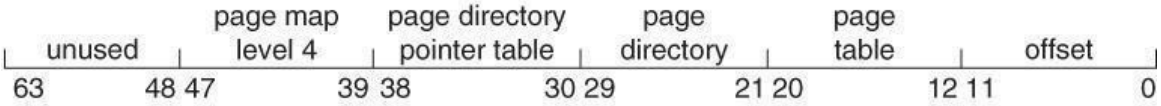
If the inner page table is currently swapped out to disk, then the page directory will have an "invalid bit" set, and the remaining 31 bits provide information on where to find the swapped out page table on the disk.



x86-64

The initial entry of Intel developing 64-bit architectures was the IA-64 (later named Itanium) architecture, but was not widely adopted.

- Meanwhile, AMD —began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set.
- The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
- Support for a 64-bit address space yields an astonishing 264 bytes of addressable memory— a number greater than 16 quintillion (or 16 exabytes).



8. IRTUAL MEMORY

- It is a technique that allows the execution of processes that may not be completely in main memory.

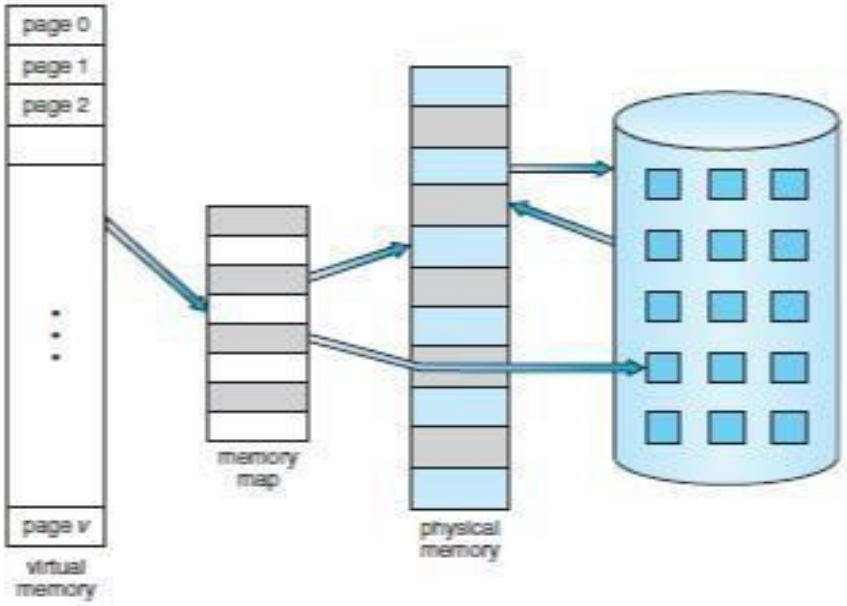
  Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

  – Only part of the program needs to be in memory for execution.

  – Logical address space can therefore be much larger than physical address space.

  – Need to allow pages to be swapped in and out.

- **Advantages:**
  - Allows the program that can be larger than the physical memory.
  - Separation of user logical memory from physical memory
  - Allows processes to easily share files & address space.
  - Allows for more efficient process creation.



- Virtual memory can be implemented using
  - Demand paging
  - Demand segmentation

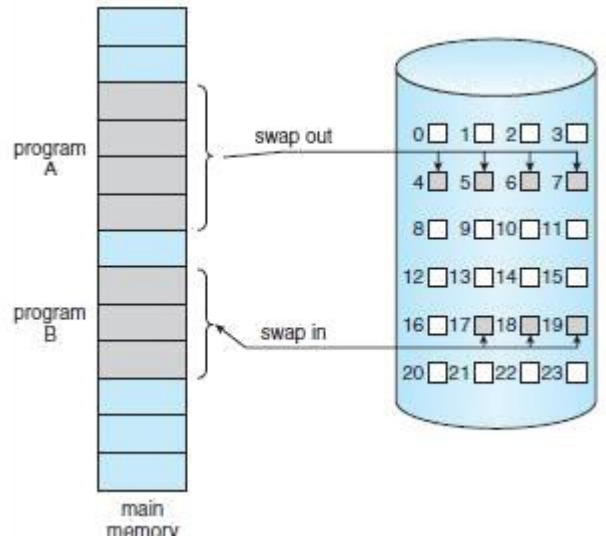
9. DEMAND PAGING

9.1 Concept

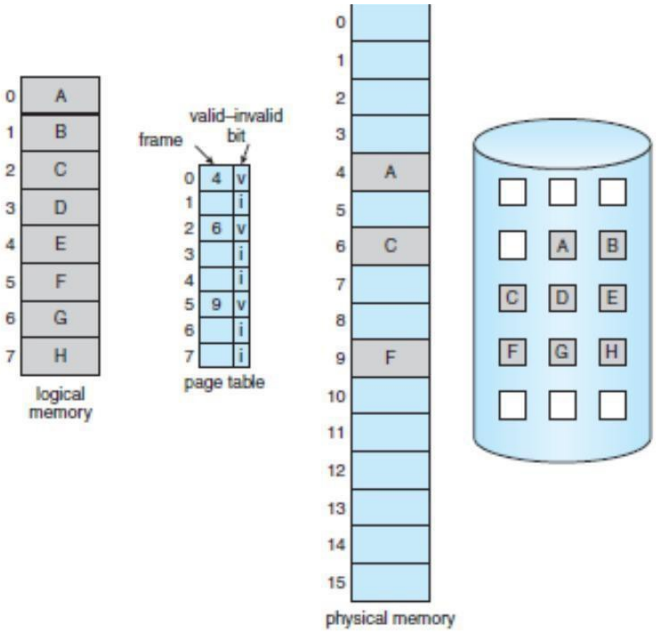
The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them (On demand). This is termed as lazy swapper.

Advantages

- Less I/O needed
- Less memory needed
- Faster response
- More users



Page table when some pages are not in main memory.



The procedure for handling this page fault

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

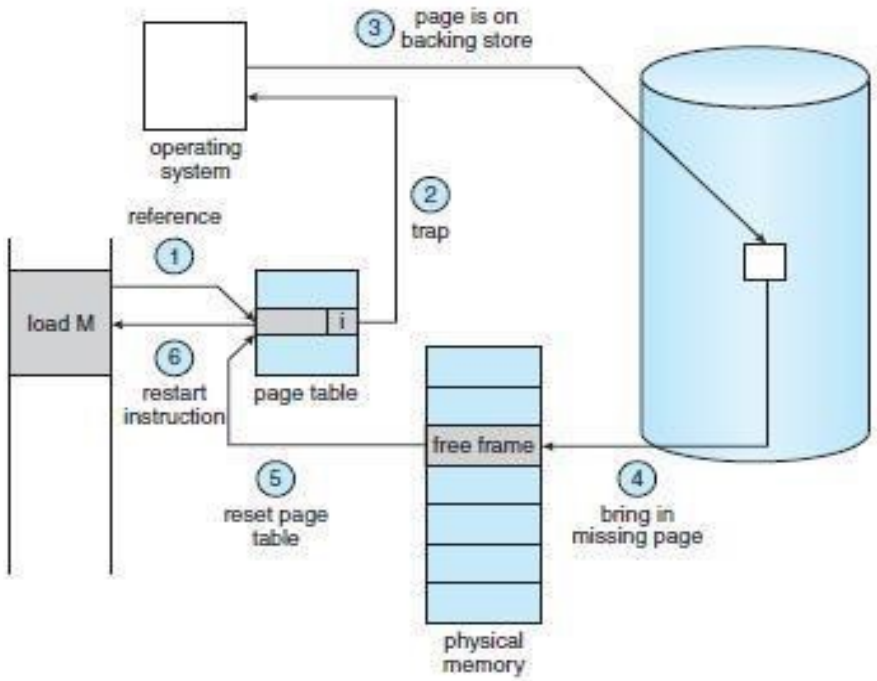
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one fromthe free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



9.2 Performance of Demand Paging

Effective Access Time (EAT) for a demand-paged memory.
Memory Access Time (ma) for most computers now ranges from 10 to 200 nanoseconds.
If there is no page fault, then EAT = ma.
If there is page fault, then

EAT = (1 – p) x (ma) + p x (page-fault time).
p: the probability of a page fault ($0 \leq p \leq 1$),
we expect p to be close to zero (a few page faults).
If p=0 then no page faults, but if p=1 then every
reference is a fault

If a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

Example

Assume an average page-fault service time of 25 milliseconds (10-3), and a Memory Access Time of 100 nanoseconds (10-9). Find the Effective Access Time?

Solution: Effective Access Time (EAT)

= (1 – p) x (ma) + p x (page fault time)
= (1 – p) x 100 + p x 25,000,000
= 100 – 100 x p + 25,000,000 x p
= 100 + 24,999,900 x p.

•Note: The Effective Access Time is directly proportional to the page-fault rate.

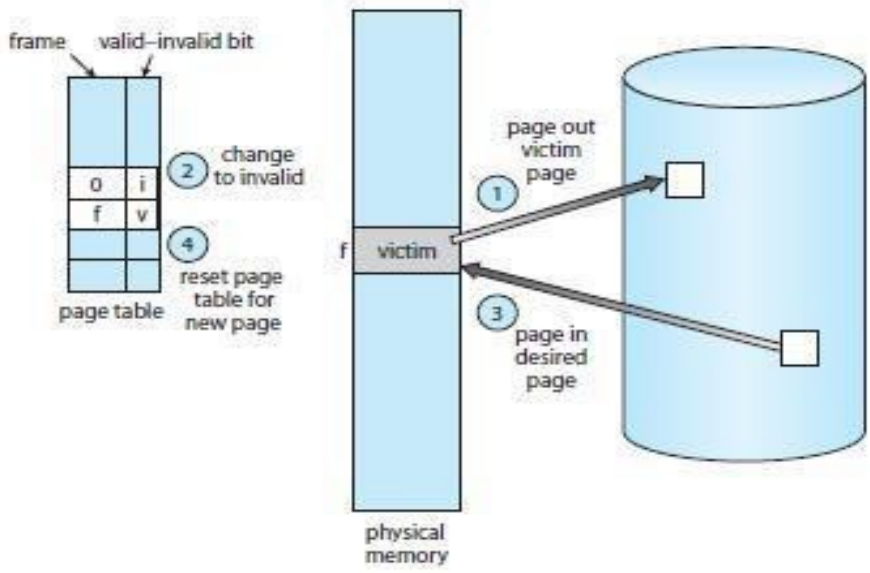
10. PAGE REPLACEMENT

10.1 Page fault

A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Need for page replacement

Page replacement is needed to decide which page needed to be replaced when new page comes in.



1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.

10.2 Page replacement algorithms

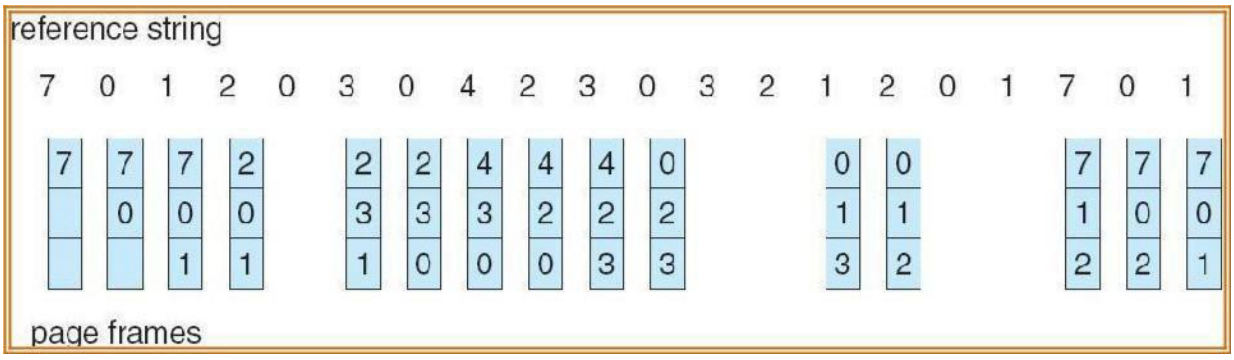
(a) FIFO page replacement algorithm

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3 (3 pages can be in memory at a time per process)

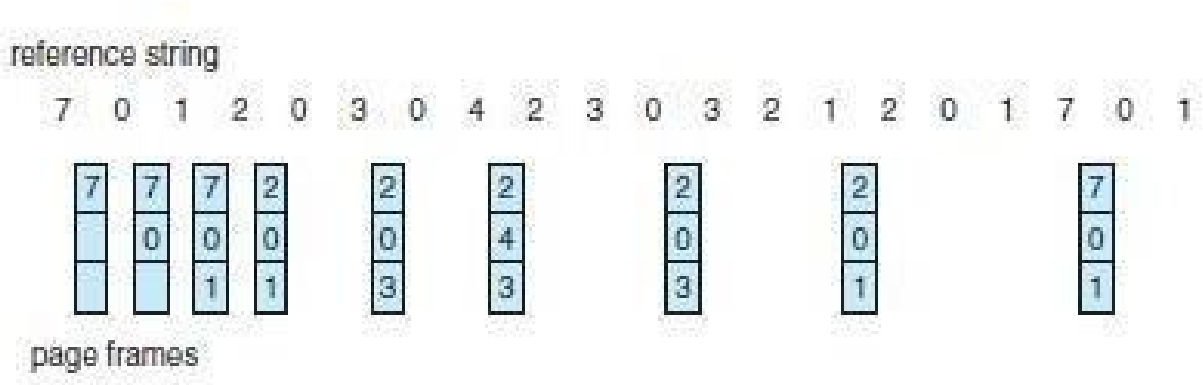


No. of page faults = 15

(b) Optimal page replacement algorithm

In this algorithm, pages are replaced which are not used for the longest duration of time in the future.

Example:



No. of page faults = 9

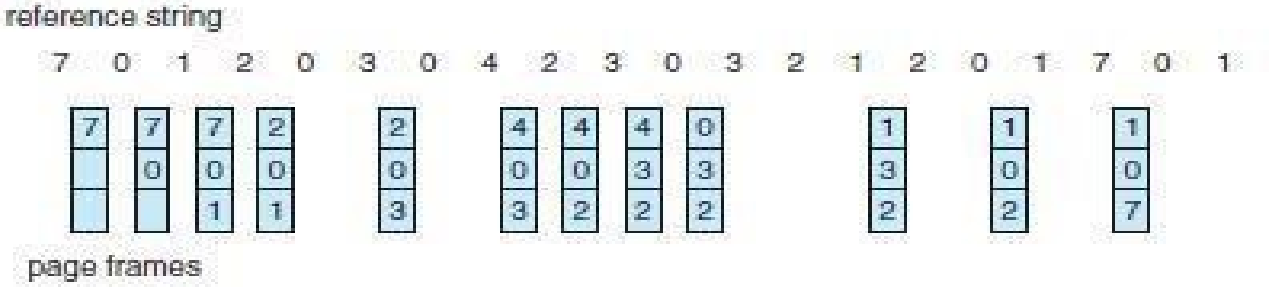
(c) LRU(Least Recently Used) page replacement algorithm

In this algorithm page will be replaced which is least recently used.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



Page Fault =12

Implementation of LRU

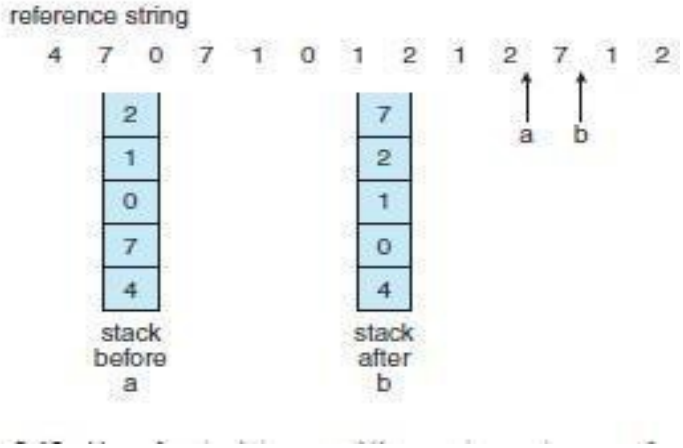
1. Counter

- The counter or clock is incremented for every memory reference.
- Each time a page is referenced , copy the counter into the time- of-use field.
- When a page needs to be replaced, replace the page with the smallest counter value.

2. Stack

- Keep a stack of page numbers
- Whenever a page is referenced, remove the page from the stack and put it on top of the stack.
- When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

Use of A Stack to Record The Most Recent Page References



(d) LRU Approximation Page Replacement

- Reference bit
  - With each page associate a reference bit, initially set to 0
  - When page is referenced, the bit is set to 1
- When a page needs to be replaced, replace the page whose reference bit is 0
- The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- Keep an 8-bit byte for each page in a table in memory.
- At regular intervals , a timer interrupt transfers control to OS.
- The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

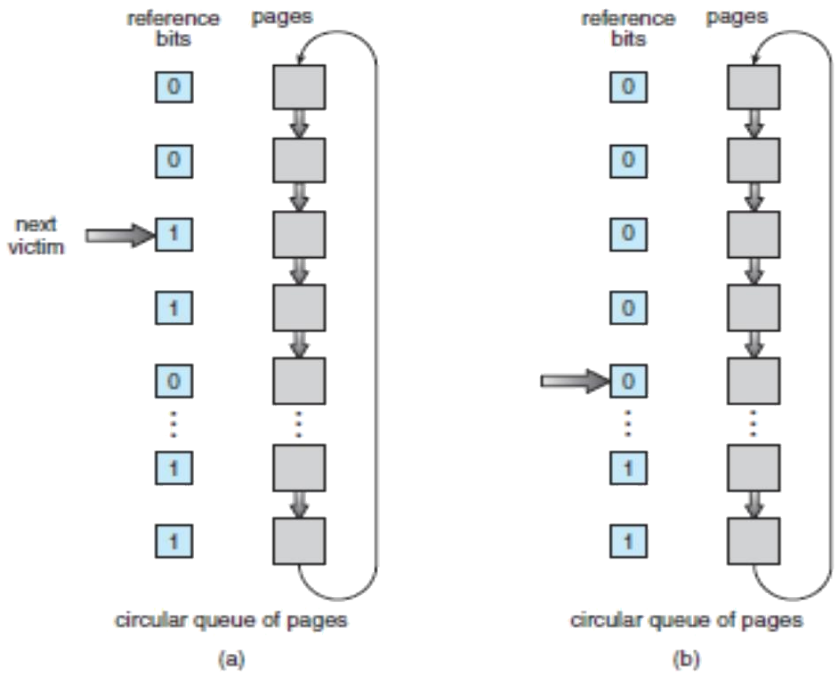
Example:

oIf reference bit is 00000000 then the page has not been used for 8 time periods.
oIf reference bit is 11111111 then the page has been used atleast once each time period. oIf the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

oBasic algorithm is FIFO
oWhen a page has been selected , check its reference bit.

- If 0 proceed to replace the page
- If 1 give the page a second chance and move on to the next FIFO page.
- When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.
- Hence a second chance page will not be replaced until all other pages are replaced.



(iii) Enhanced Second Chance Algorithm

o Consider both reference bit and modify bit o

There are four possible classes

1. (0,0) – neither recently used nor modified est page to replace
2. (0,1) – not recently used but modifiedpage has to be written out before replacement.
3. (1,0) - recently used but not modified page may be used again
4. (1,1) – recently used and modifiedpage may be used again and page has to be written to disk

(iv) Counting-Based Page Replacement

o Keep a counter of the number of references that have been made to each page

1. **Least Frequently Used (LFU)Algorithm**: replaces page with smallest count
2. **Most Frequently Used (MFU)Algorithm**: replaces page with largest count
   - □ is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

11. ALLOCATION OF FRAMES

11.1 Allocation of Frames

o There are two major allocation schemes

- □ Equal Allocation
- □ Proportional Allocation

Equal allocation

- If there are n processes and m frames then allocate m/n frames to each process.
- **Example:** If there are 5 processes and 100 frames, give each process 20 frames.
- Allocate according to the size of process

Let si be the size of process i.

Let m be the total no. of

frames Then $S = \sum si$

$ai = si / S * m$

where ai is the no.of frames allocated to process i.

11.2 Global vs. Local Replacement

- **Global replacement** – each process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local replacement** – each process selects from only its own set of allocated frames.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames

$10/137 \times 62 \approx 4$, and

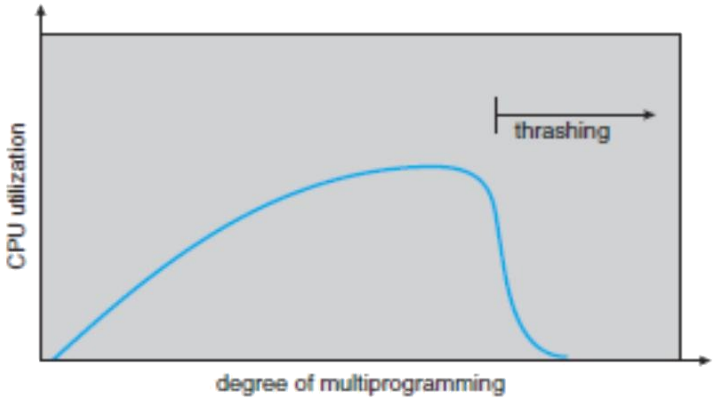
$127/137 \times 62 \approx 57$.

12. THRASHING

Thrashing

- High paging activity is called **thrashing**.
- If a process does not have enough pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process is added to the system
- When the CPU utilization is low, the OS increases the degree of multiprogramming.
- If global replacement is used then as processes enter the main memory they tend to steal frames belonging to other processes.
- Eventually all processes will not have enough frames and hence the page fault rate becomes very high.

o Thus swapping in and swapping out of pages only takes place.
o This is the cause of thrashing.

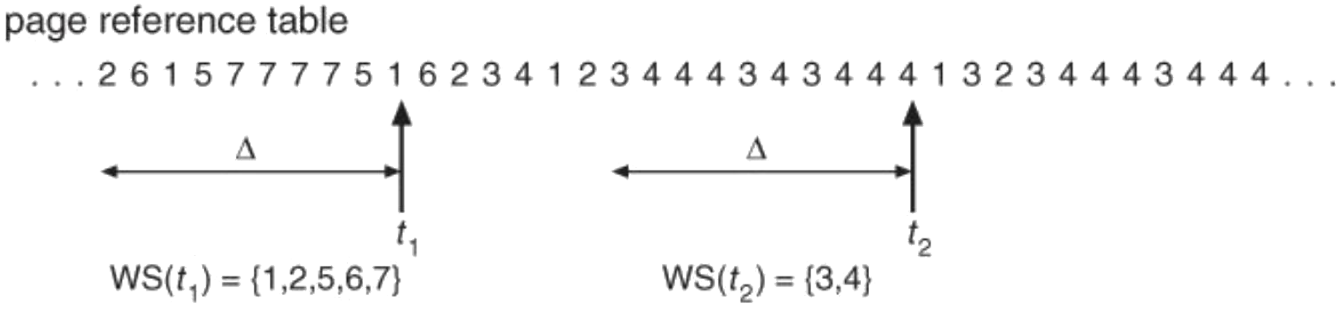


o To **limit thrashing**, we can use a **local replacement** algorithm. o To prevent thrashing, there are two methods namely ,

- Working Set Strategy
- Page Fault Frequency

1. Working-Set Strategy

o It is based on the assumption of the model of locality.
o Locality is defined as the set of pages actively used together.
o Whatever pages are included in the most recent page references are said to be in the processes working set window, and comprise its current working set .



If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set time units after its last reference.

- Thus, the working set is an approximation of the program's locality.
- if Δ = 10 memory references, then the working set at time t1 is {1, 2, 5,6, 7}.
- By time t2, the working set has changed to {3, 4}.
- The accuracy of the working set depends on the selection of .
- If Δ is too small, it will not encompass the entire locality; if is too large, it may overlap several localities.
- In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

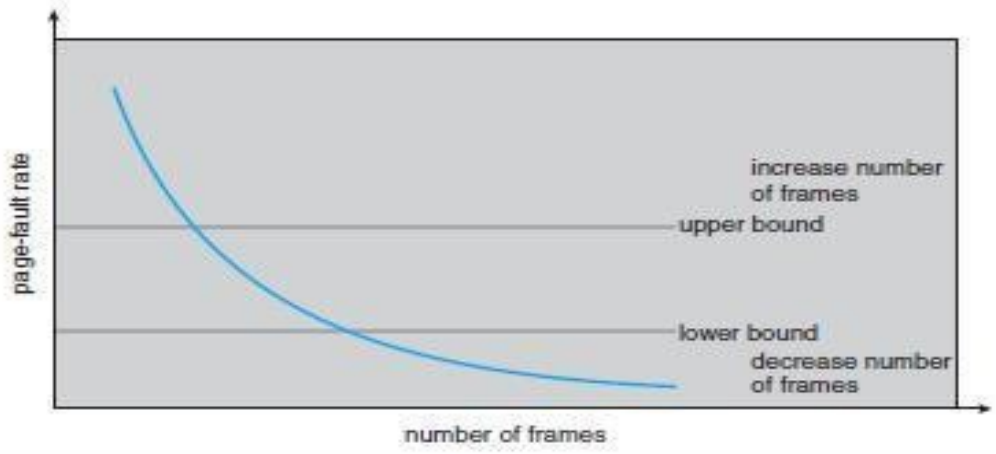
 The most important property of the working set, then, is its size.
 If we compute the working-set size, WSSi , for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

 where D is the total demand for frames. Each process is actively using the pages in its working set.
 Thus, process i needs WSSi frames. If the total demand is greater than the total number of available frames (D> m), thrashing will occur, because some processes will not have enough frames.

2. Page-Fault Frequency Scheme

- Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate.
- When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
  We can establish upper and lower bounds on the desired page-fault rate.
- If the actual page-fault rate exceeds the upper limit, we allocate the process another frame.
- If the page-fault rate falls below the lower limit, we remove a frame from the process.
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.



13. ALLOCATING KERNEL MEMORY

Allocating Kernel Memory

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm such as those discussed in Section 9.4 and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that if a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

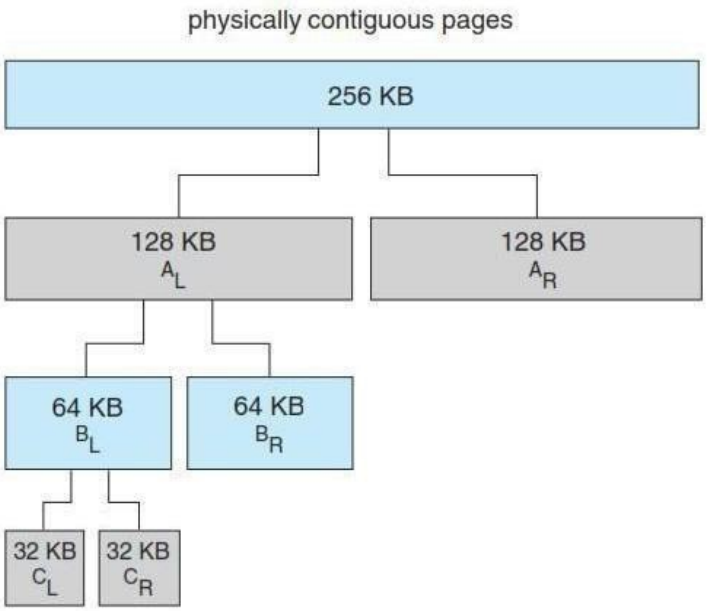
Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory— without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

Buddy System

The buddy system allocates memory from a fixed -size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4KB, 8KB, 16KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16K segment



Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory.

The segment is initially divided into two buddies—which we will call AL and AR—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies— BL and BR.

However, the next-highest power of 2 from 21 KB is 32 KB so either BL or BR is again divided into two 32-KB buddies, CL and CR. One of these buddies is used to satisfy the 21-KB request.

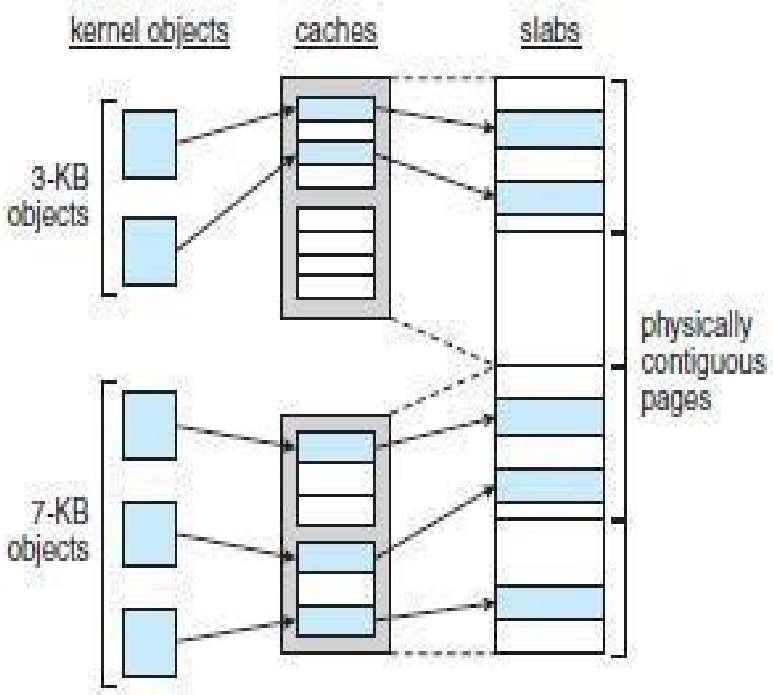
Slab Allocation

A second strategy for allocating kernel memory is known as slab allocation. A slab is made up of one or more physically contiguous pages. A cache consists of one or more slabs.

The slab-allocation algorithm uses caches to store kernel objects.

When a cache is created, a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.

For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects.



In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.
2. Empty. All objects in the slab are marked as free.
3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab.

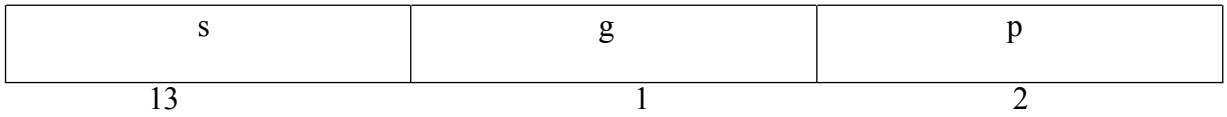
If none exists, a free object is assigned from an empty slab.

If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

14. SEGMENTATION WITH PAGING

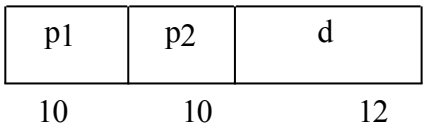
- The IBM OS/ 2.32 bit version is an operating system running on top of the Intel 386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.
- The local-address space of a process is divided into two partitions.
  The first partition consists of up to 8 KB segments that are private to that process.
  The second partition consists of up to 8KB segments that are shared among all the processes.
- Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**.
- Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.

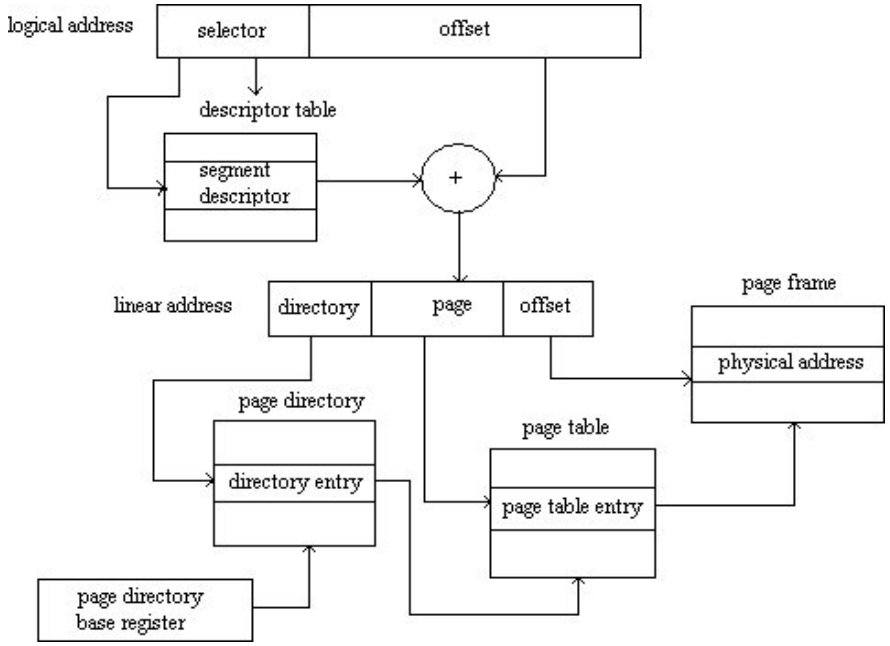
The logical address is a pair (selector, offset) where the selector is a16-bit number:



Where s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

- The base and limit information about the segment in question are used to generate a linear-address.
- First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.
- The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows.





oTo improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

oIf the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.