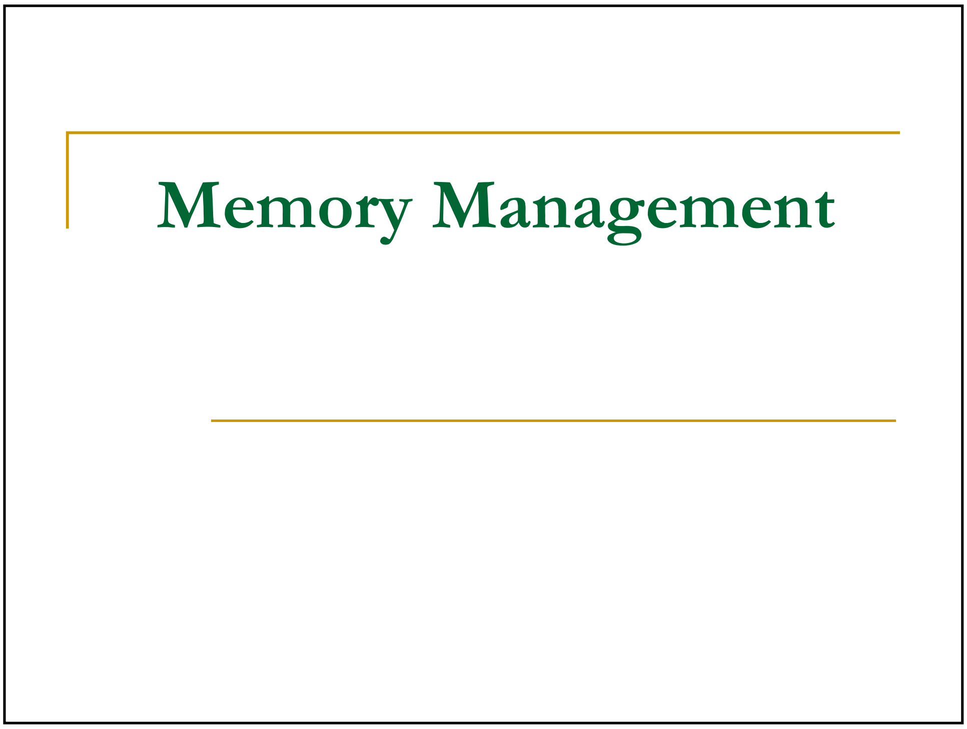
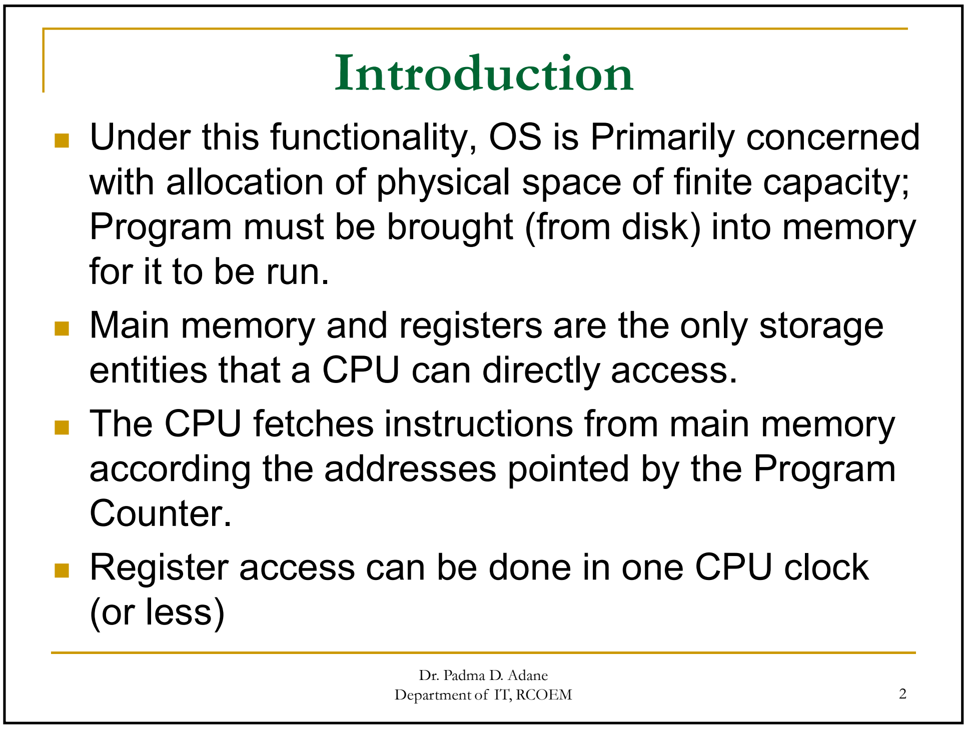
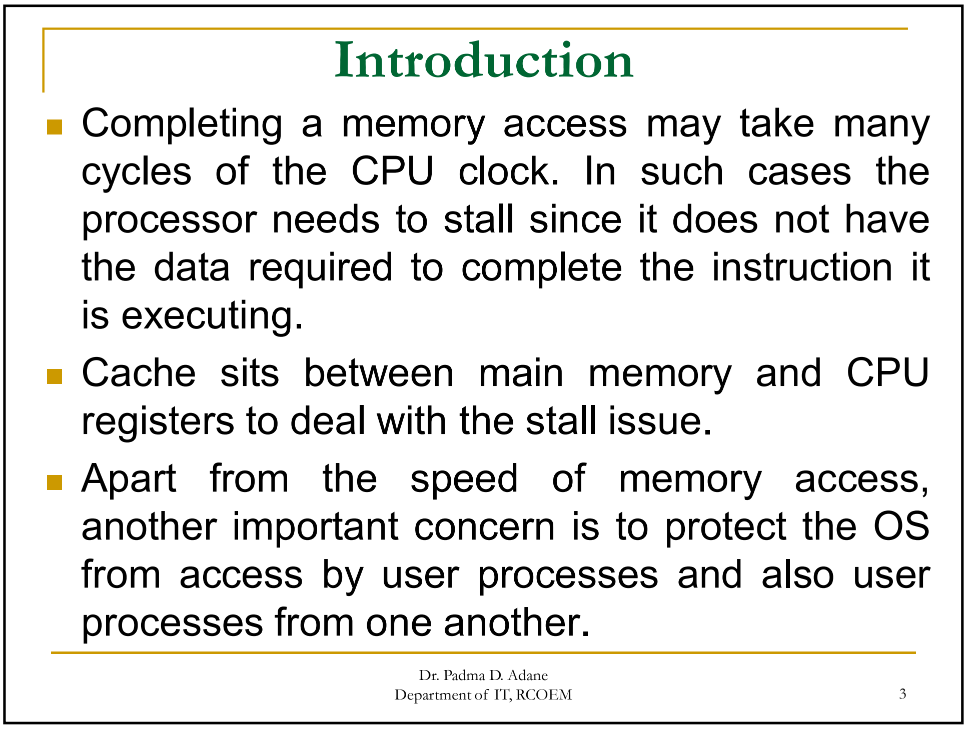


Memory Management



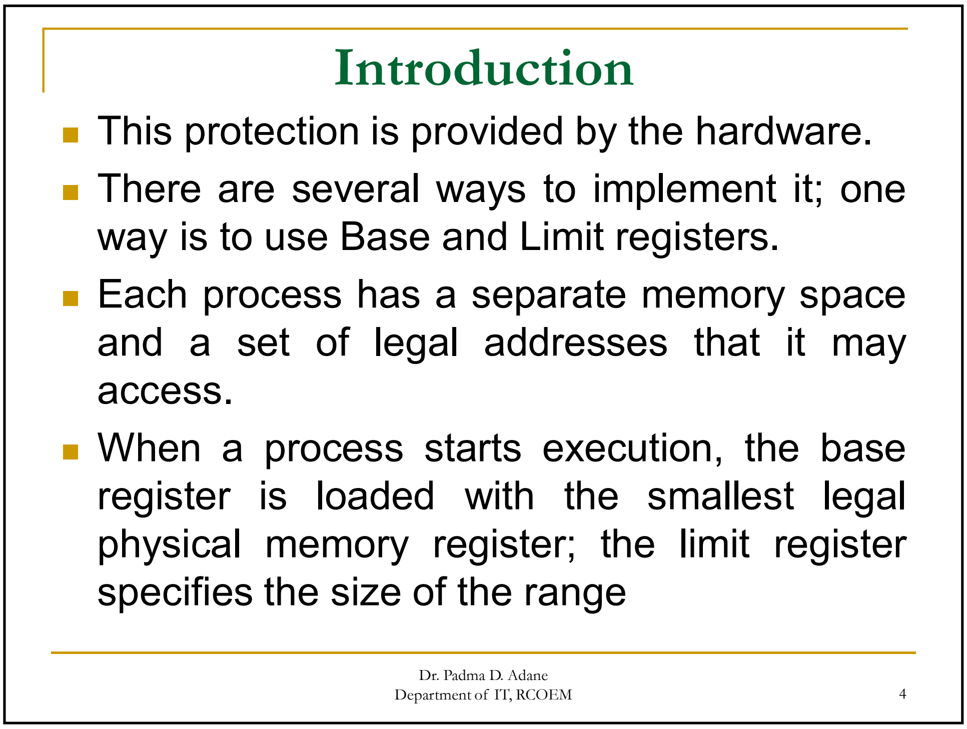
Introduction

- Under this functionality, OS is Primarily concerned with allocation of physical space of finite capacity; Program must be brought (from disk) into memory for it to be run.
- Main memory and registers are the only storage entities that a CPU can directly access.
- The CPU fetches instructions from main memory according the addresses pointed by the Program Counter.
- Register access can be done in one CPU clock (or less)



Introduction

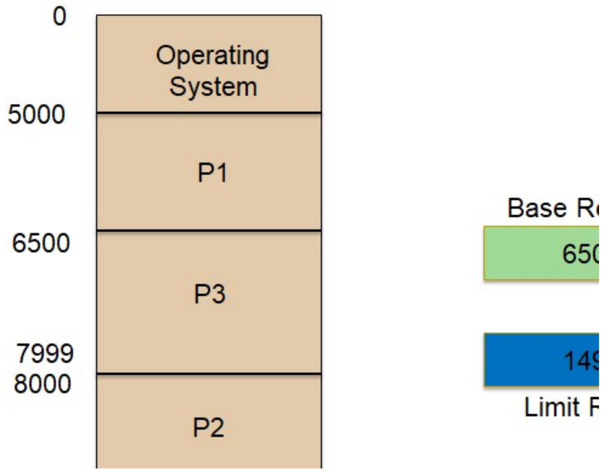
- Completing a memory access may take many cycles of the CPU clock. In such cases the processor needs to stall since it does not have the data required to complete the instruction it is executing.
- Cache sits between main memory and CPU registers to deal with the stall issue.
- Apart from the speed of memory access, another important concern is to protect the OS from access by user processes and also user processes from one another.



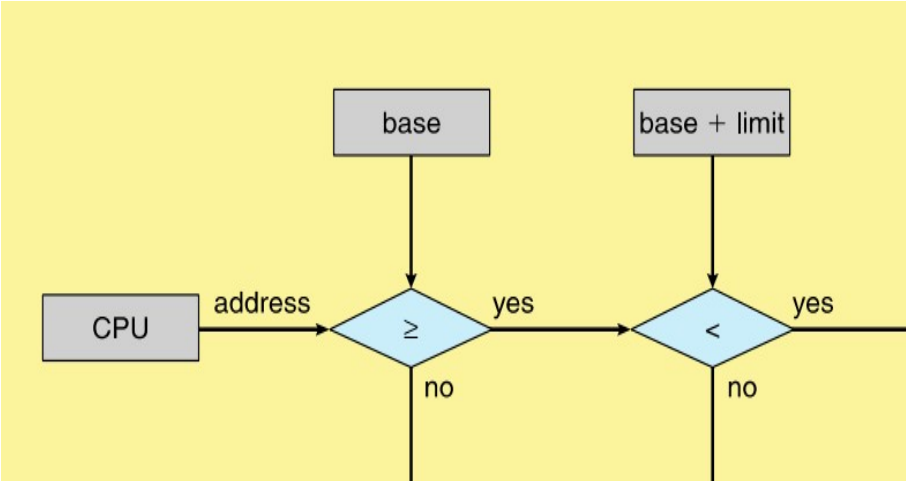
Introduction

- This protection is provided by the hardware.
- There are several ways to implement it; one way is to use Base and Limit registers.
- Each process has a separate memory space and a set of legal addresses that it may access.
- When a process starts execution, the base register is loaded with the smallest legal physical memory register; the limit register specifies the size of the range

Introduction



Introduction

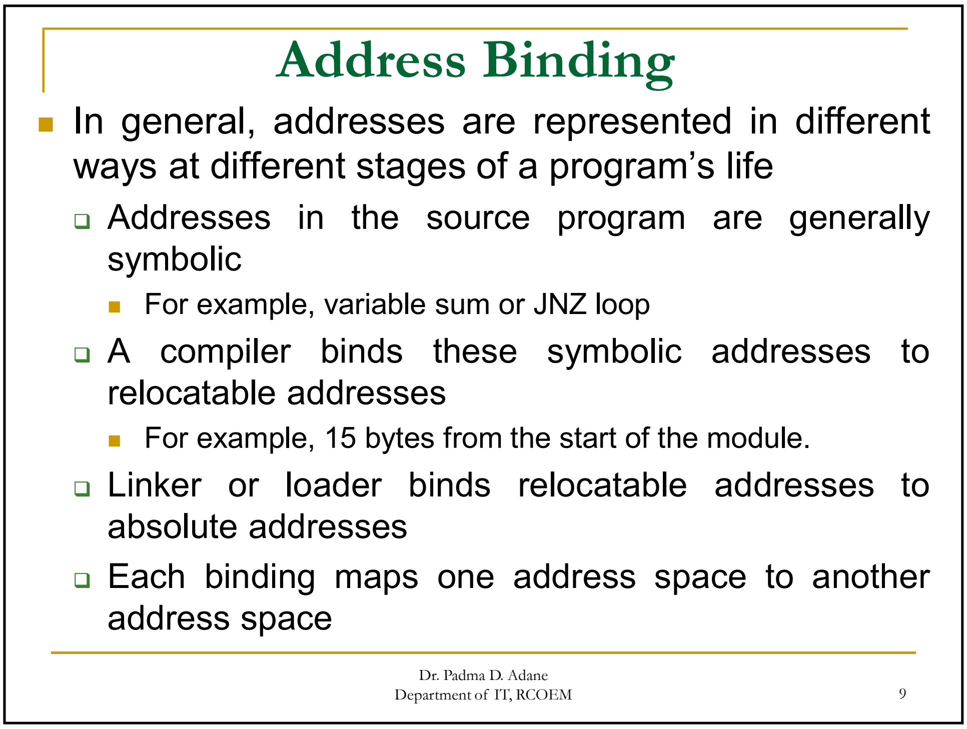


Introduction

- CPU must check that every memory access generated in user mode is between the base and base + limit for that user.
- This scheme prevents a user program from (accidently or deliberately) modifying the code or data structures of either the operating system or other users.
- The base and limit registers can be loaded only by the OS, which uses a privileged instruction to do so.

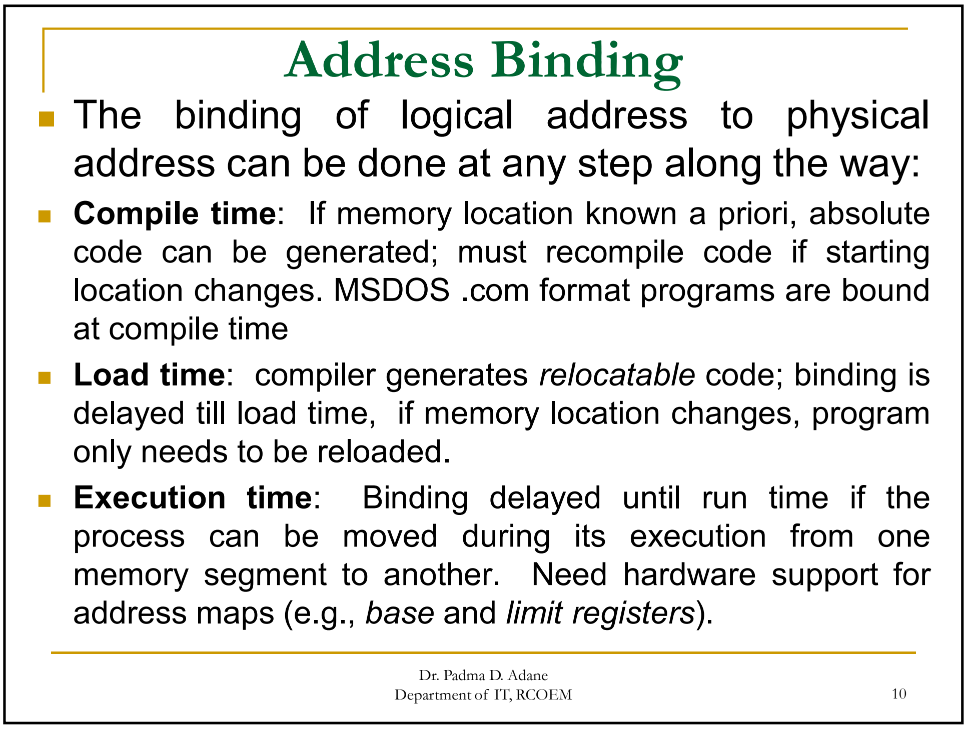
Address Binding

- When a program is brought into memory to run, it is not know a priori where it is going to reside in physical memory. Therefore, it is convenient to assume that the first address of the program starts at 0000.
- However, it is not practical to have first physical address of user program to always start at 0000.
- Computer systems provide some hardware/ software support to handle this aspect of memory management.



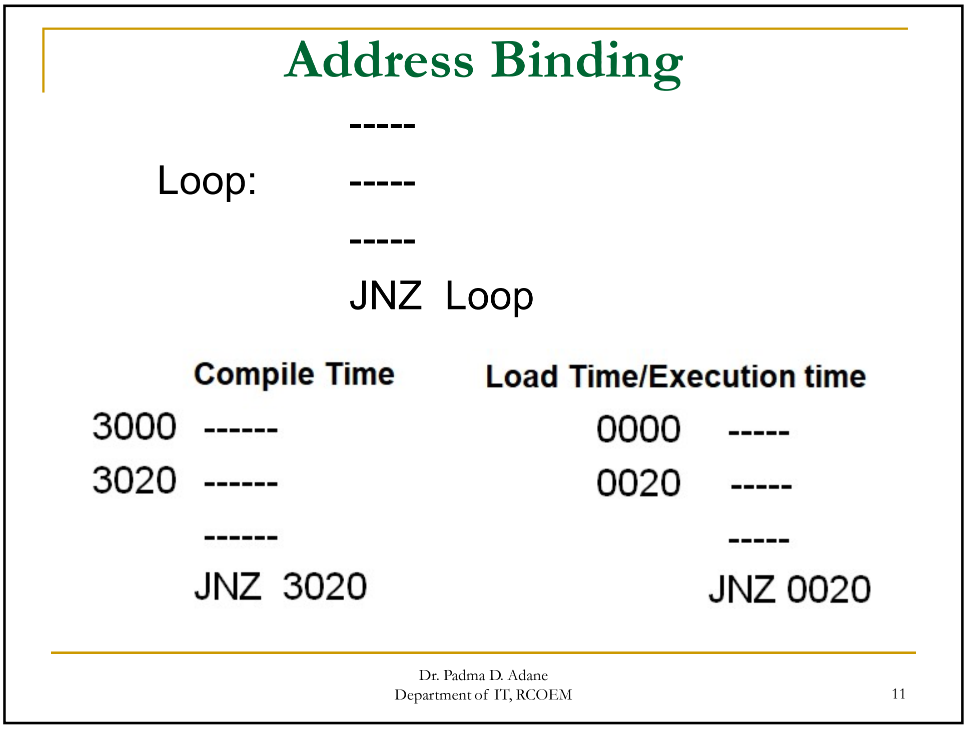
Address Binding

- In general, addresses are represented in different ways at different stages of a program's life
  - Addresses in the source program are generally symbolic
    - For example, variable sum or JNZ loop
  - A compiler binds these symbolic addresses to relocatable addresses
    - For example, 15 bytes from the start of the module.
  - Linker or loader binds relocatable addresses to absolute addresses
  - Each binding maps one address space to another address space



Address Binding

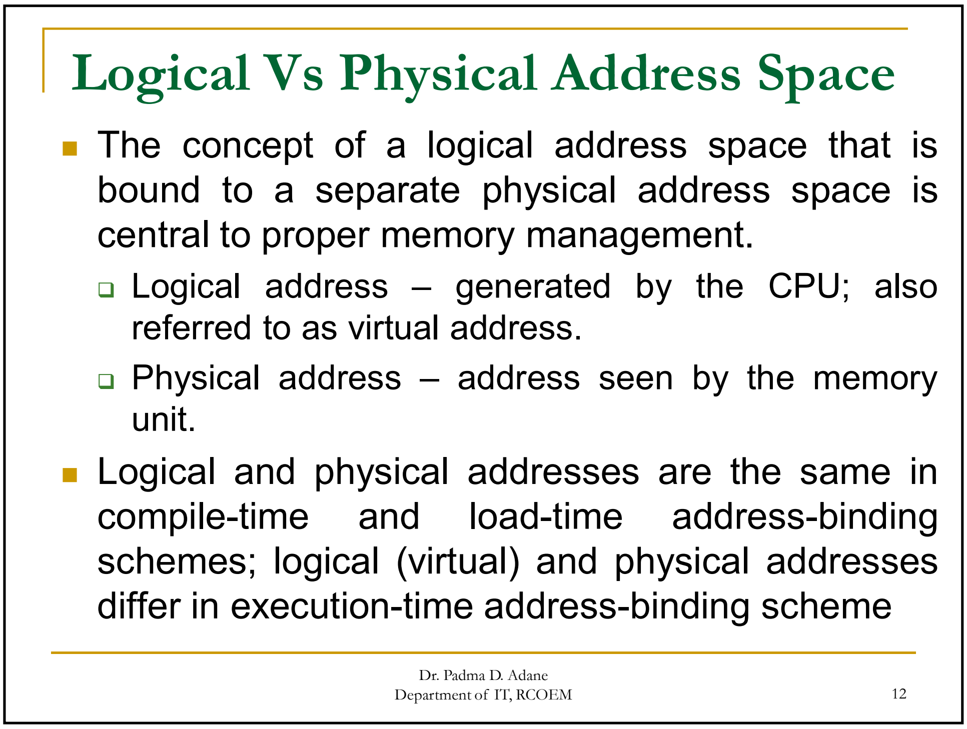
- The binding of logical address to physical address can be done at any step along the way:
- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes. MSDOS .com format programs are bound at compile time
- **Load time**: compiler generates *relocatable* code; binding is delayed till load time, if memory location changes, program only needs to be reloaded.
- **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*).



Address Binding

```
            -----
Loop:       -----
            -----
            JNZ  Loop
```

Compile Time		Load Time/Execution time	
3000	------	0000	-----
3020	------	0020	-----
	------		-----
	JNZ 3020		JNZ 0020

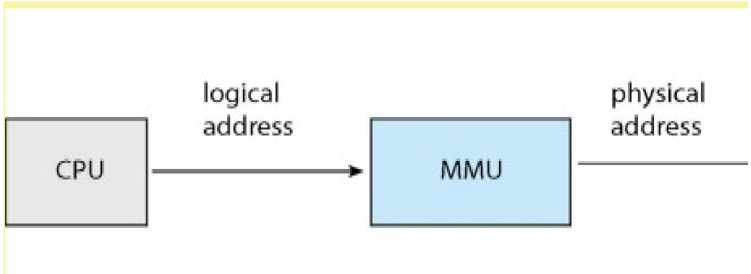


Logical Vs Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
  - 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)

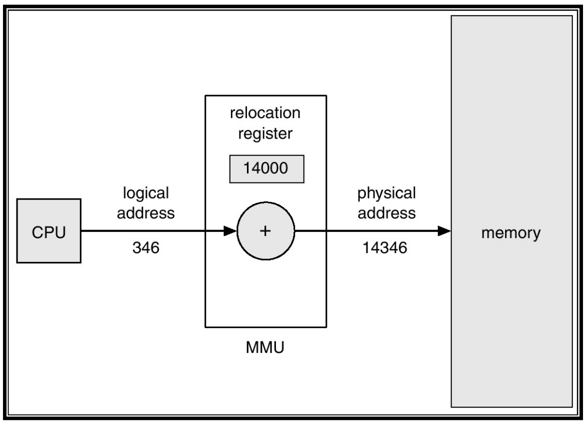
- Hardware device that maps logical (virtual) address to physical address.

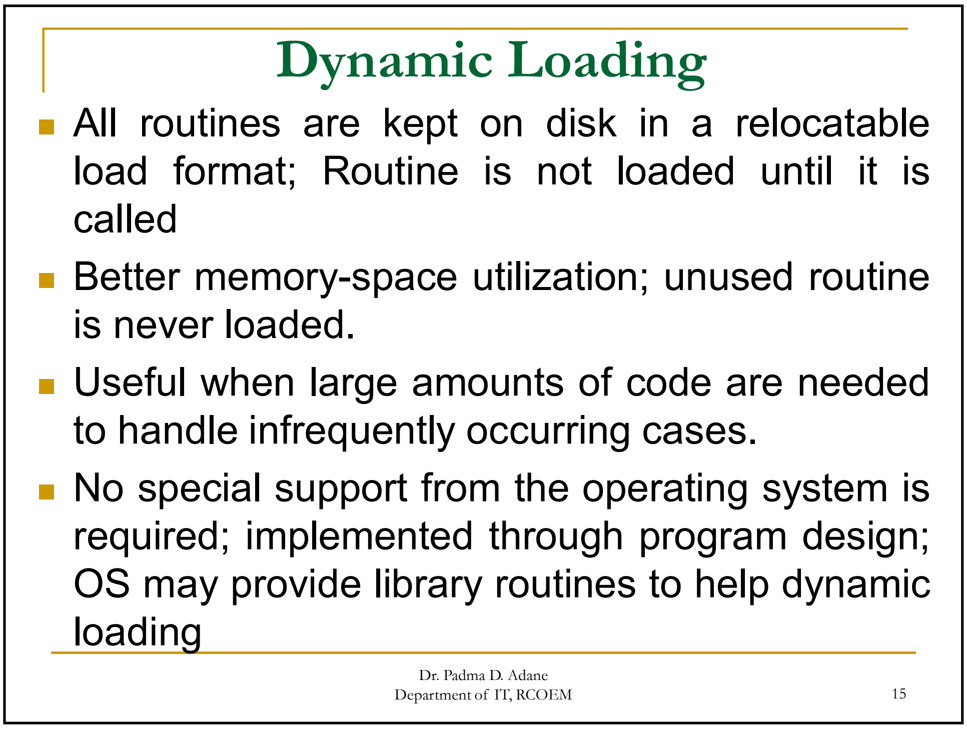


- The user program deals with logical addresses; it never sees the real physical addresses.

Dynamic Relocation using Relocation Register

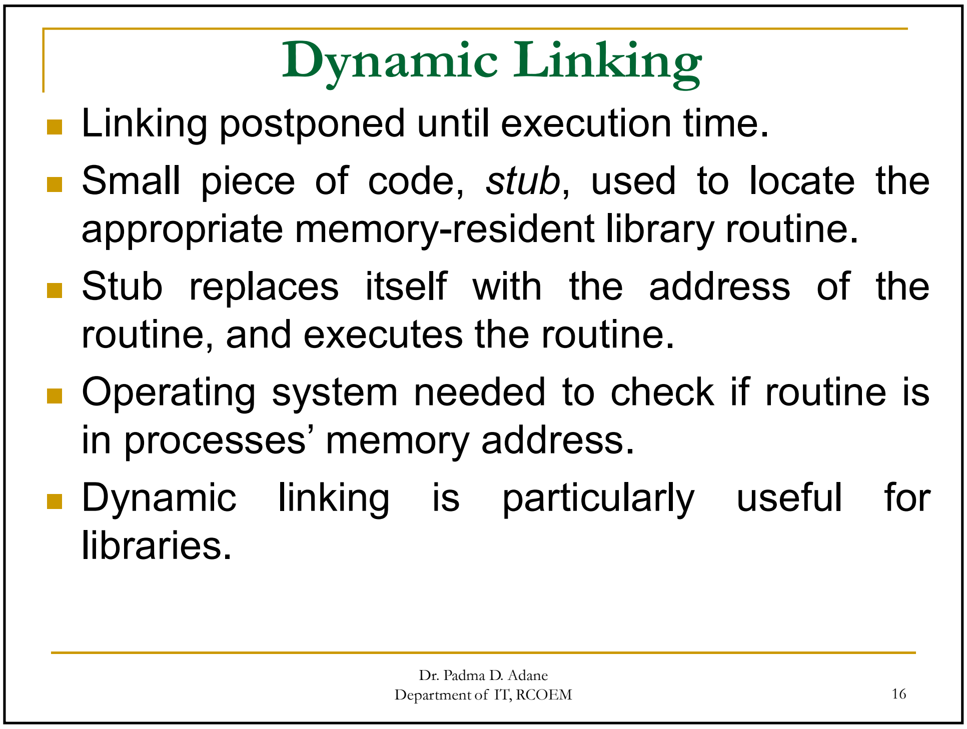
- The value in the relocation register is added to every logical address generated by a program





Dynamic Loading

- All routines are kept on disk in a relocatable load format; 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; OS may provide library routines to help dynamic loading



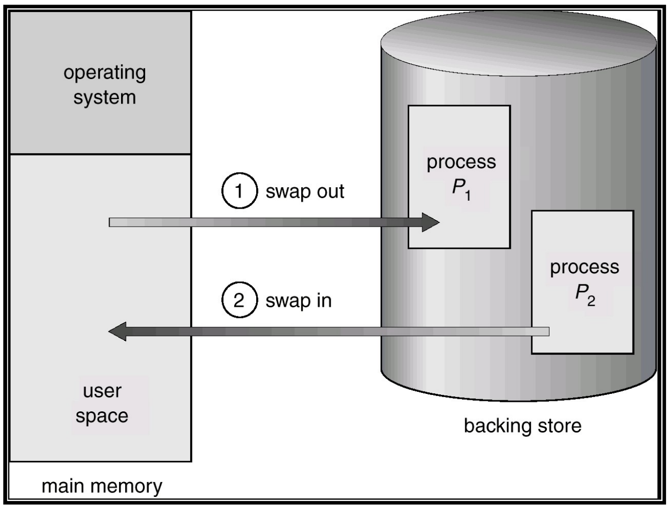
Dynamic Linking

- 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.

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.

Swapping

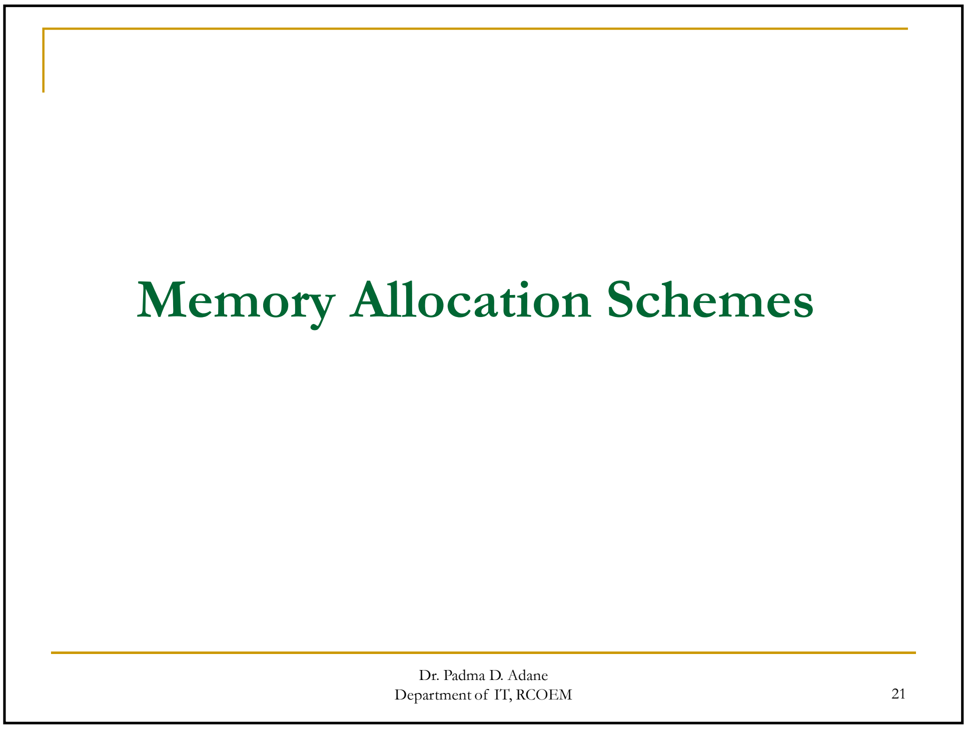


Swapping Considerations

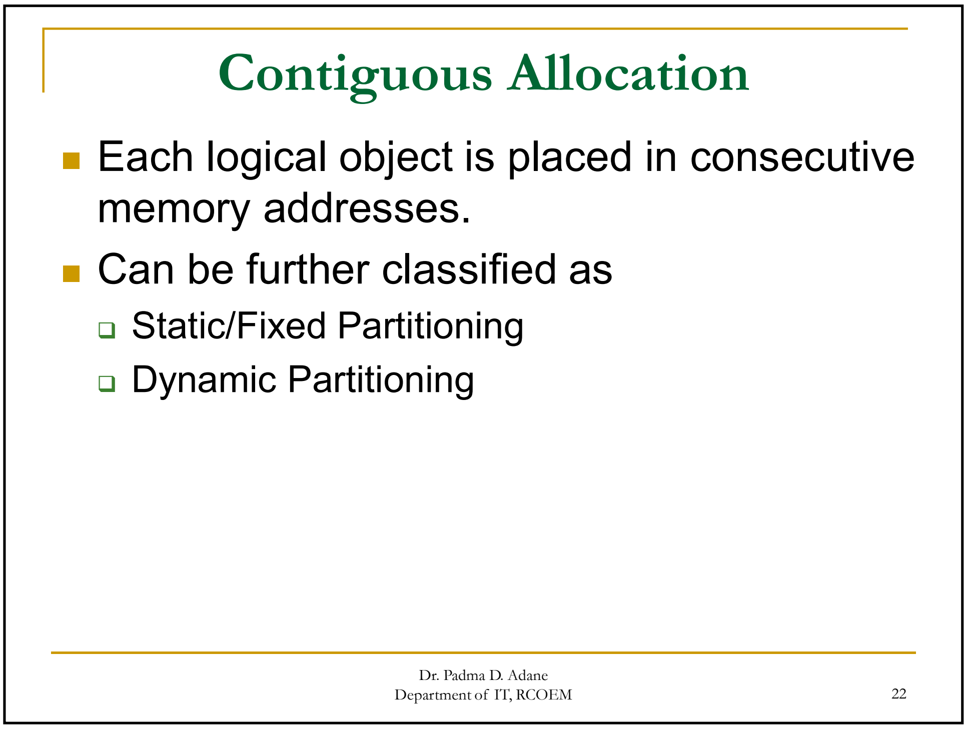
- It depends on address binding, where in memory, a swapped in process will be brought back; for compile and load time binding, the process needs to brought back into same locations from where it was swapped out. For execution time binding, it can be brought into a different location.
- To swap out a process, the backing store must have considerable space; the swapped out image of a process might be different from the static image when it was loaded into the memory.

Swapping Considerations

- A swapped out process must not have any pending I/O operations; it might be accessing user memory for I/O buffers. Two approaches to handle this situation:
  - Never swap a process with pending I/O
  - Execute I/O operations only into OS buffers
- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows; swapping is normally disabled, it is started when number of processes are more and are using a threshold amount of memory

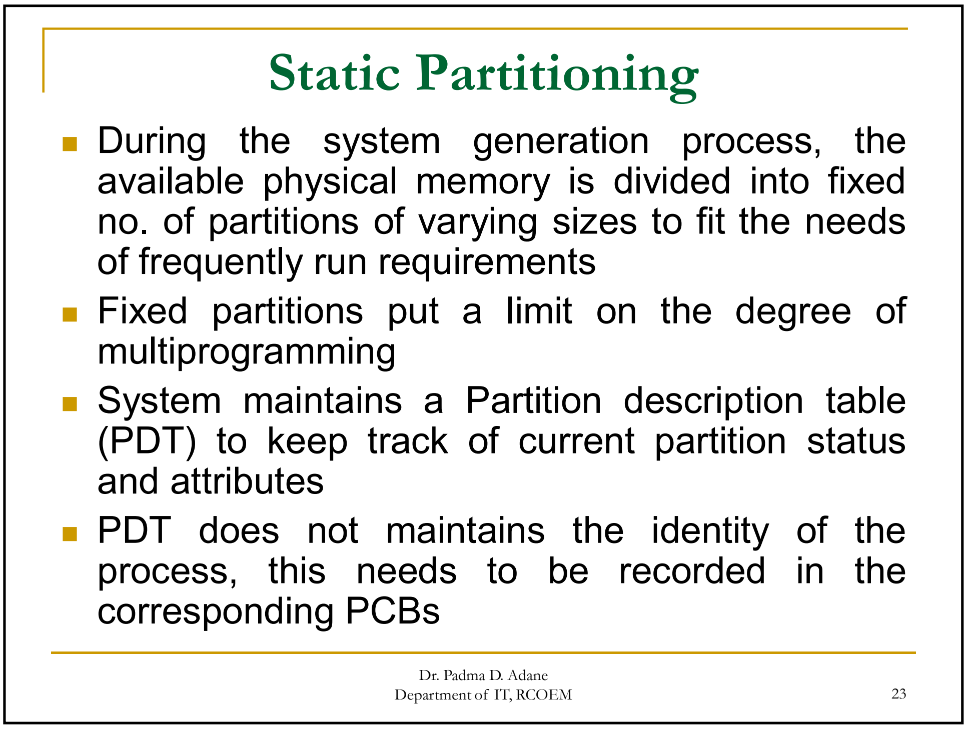


Memory Allocation Schemes



Contiguous Allocation

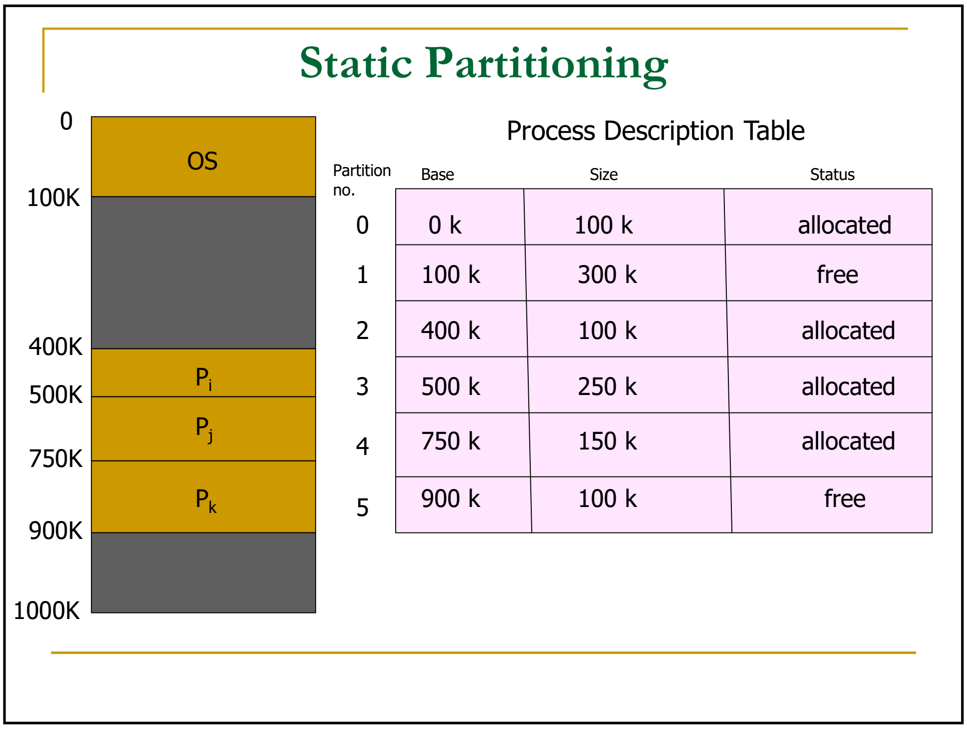
- Each logical object is placed in consecutive memory addresses.
- Can be further classified as
  - Static/Fixed Partitioning
  - Dynamic Partitioning



Static Partitioning

- During the system generation process, the available physical memory is divided into fixed no. of partitions of varying sizes to fit the needs of frequently run requirements
- Fixed partitions put a limit on the degree of multiprogramming
- System maintains a Partition description table (PDT) to keep track of current partition status and attributes
- PDT does not maintains the identity of the process, this needs to be recorded in the corresponding PCBs

Dr. Padma D. Adane
Department of IT, RCOEM 23



Static Partitioning

Process Description Table

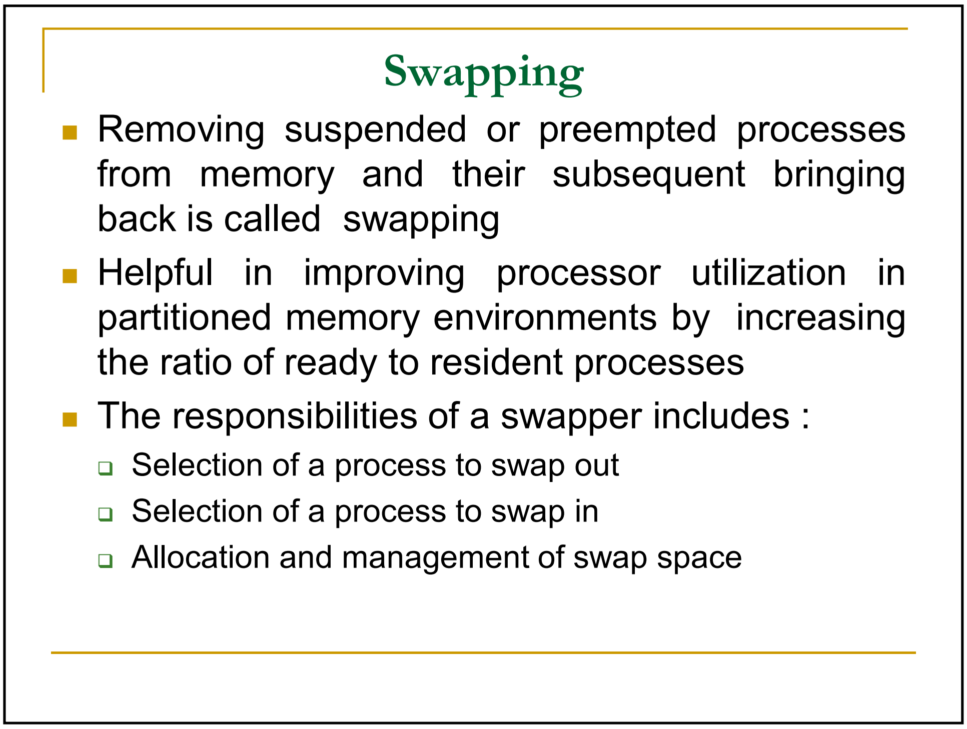
Partition no.	Base	Size	Status
0	0 k	100 k	allocated
1	100 k	300 k	free
2	400 k	100 k	allocated
3	500 k	250 k	allocated
4	750 k	150 k	allocated
5	900 k	100 k	free

Static Partitioning

- **There is a close relationship and interaction between memory management and scheduling functions of the OS.**
- By influencing the membership of the set of resident processes, a memory manager may affect the scheduler's ability to perform; on the other hand, the effectiveness of the short term scheduler influences the memory manager by affecting the average memory residence times

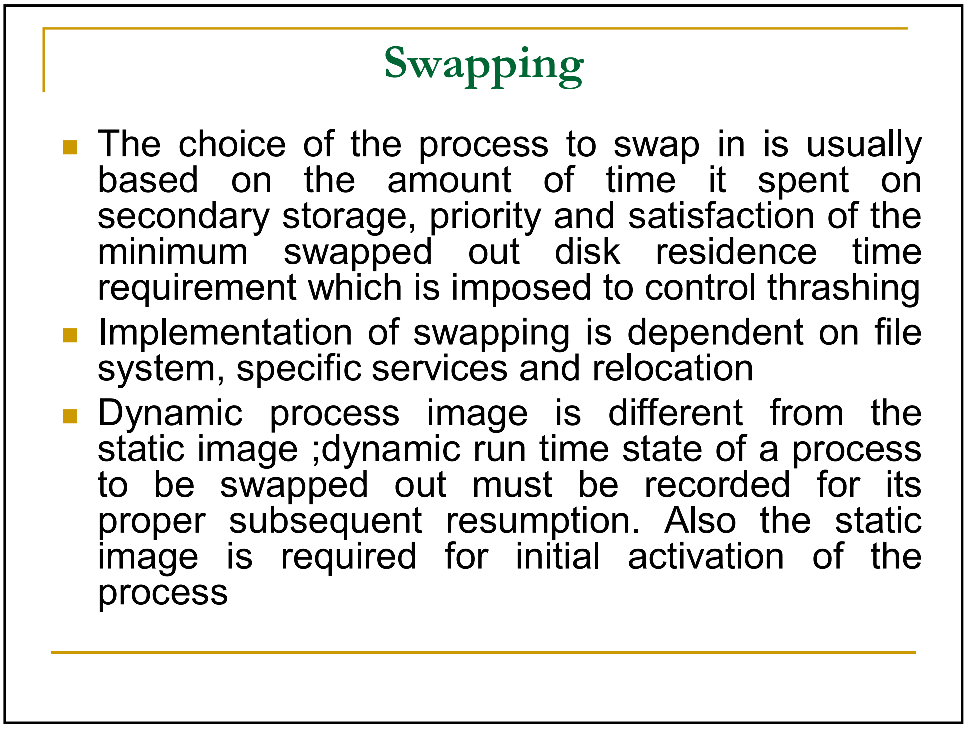
Fragmentation

- Memory fragmentation refers to the unused/free memory that cannot be assigned to any process
- Static partitioning results in Internal fragmentation.
- The unused memory that is part of a partition allocated to a process whose size is smaller than the allocated partition.



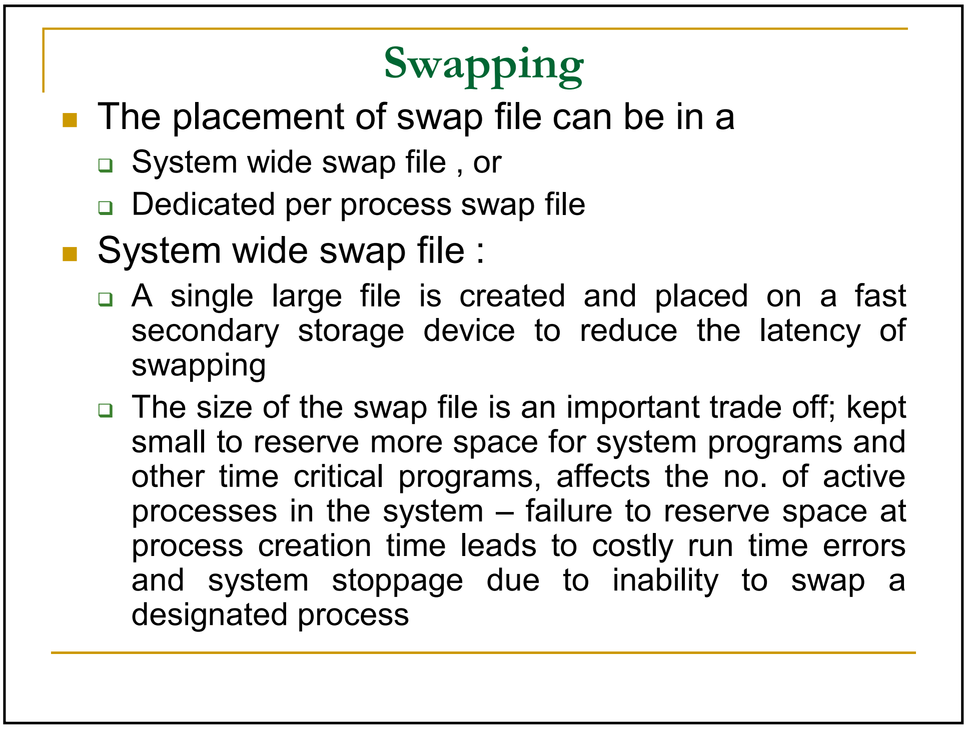
Swapping

- Removing suspended or preempted processes from memory and their subsequent bringing back is called swapping
- Helpful in improving processor utilization in partitioned memory environments by increasing the ratio of ready to resident processes
- The responsibilities of a swapper includes :
  - Selection of a process to swap out
  - Selection of a process to swap in
  - Allocation and management of swap space



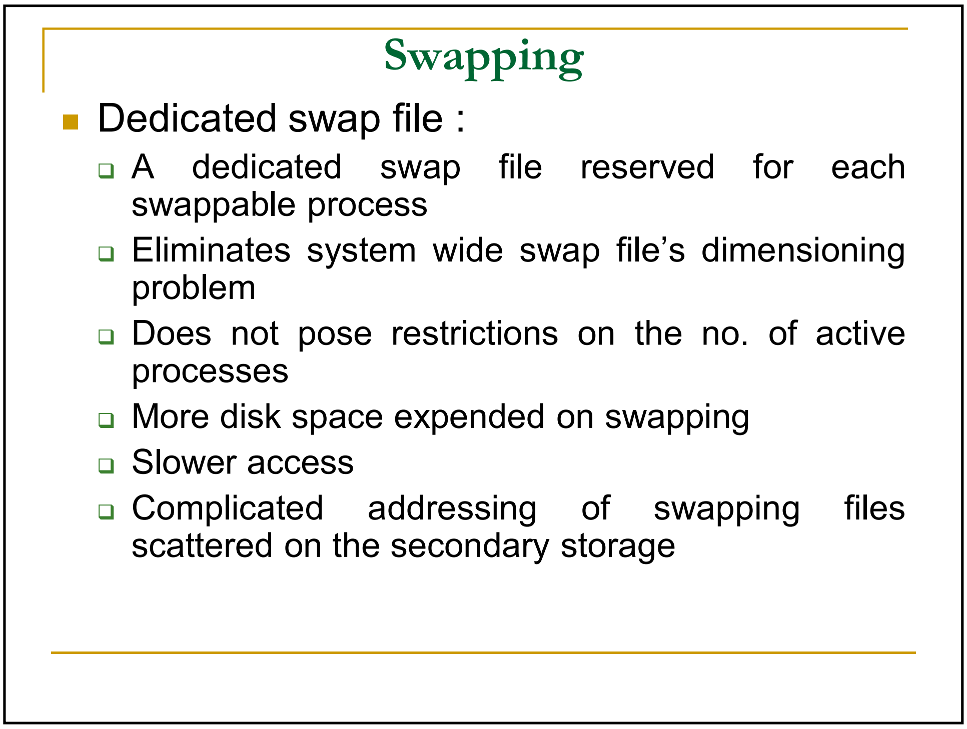
Swapping

- The choice of the process to swap in is usually based on the amount of time it spent on secondary storage, priority and satisfaction of the minimum swapped out disk residence time requirement which is imposed to control thrashing
- Implementation of swapping is dependent on file system, specific services and relocation
- Dynamic process image is different from the static image ;dynamic run time state of a process to be swapped out must be recorded for its proper subsequent resumption. Also the static image is required for initial activation of the process



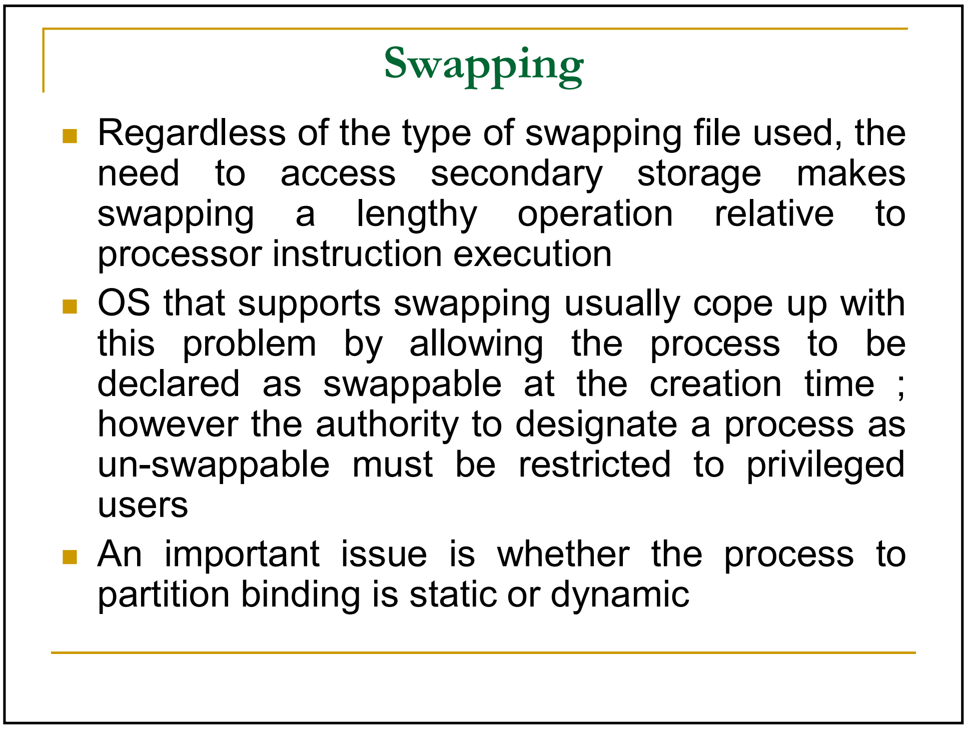
Swapping

- The placement of swap file can be in a
  - System wide swap file , or
  - Dedicated per process swap file
- System wide swap file :
  - A single large file is created and placed on a fast secondary storage device to reduce the latency of swapping
  - The size of the swap file is an important trade off; kept small to reserve more space for system programs and other time critical programs, affects the no. of active processes in the system – failure to reserve space at process creation time leads to costly run time errors and system stoppage due to inability to swap a designated process



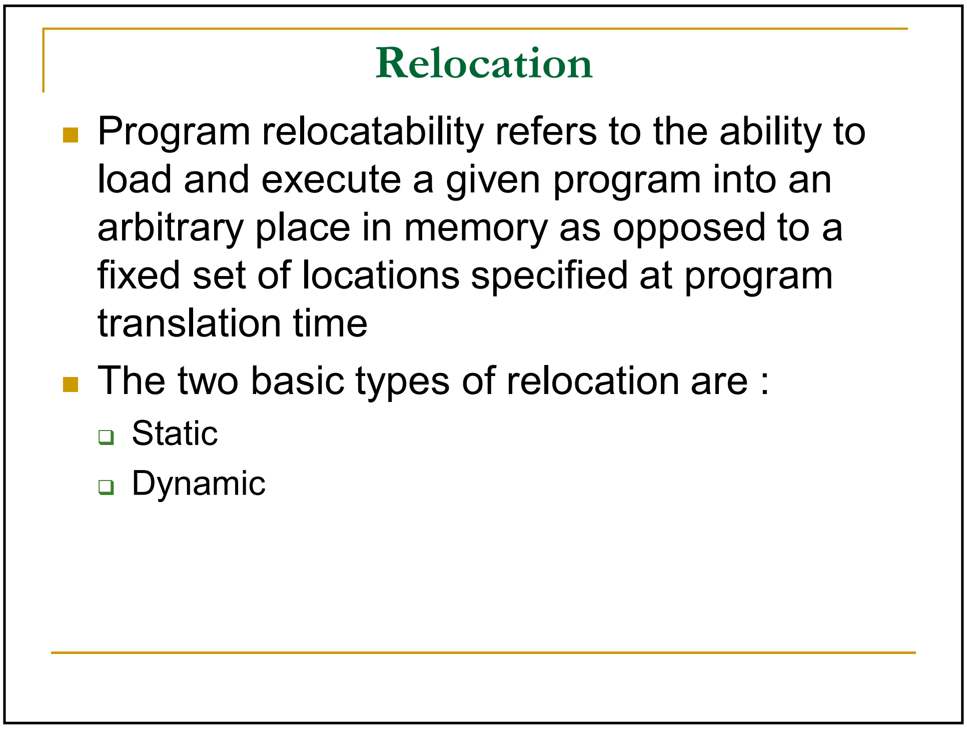
Swapping

- Dedicated swap file :
  - A dedicated swap file reserved for each swappable process
  - Eliminates system wide swap file’s dimensioning problem
  - Does not pose restrictions on the no. of active processes
  - More disk space expended on swapping
  - Slower access
  - Complicated addressing of swapping files scattered on the secondary storage



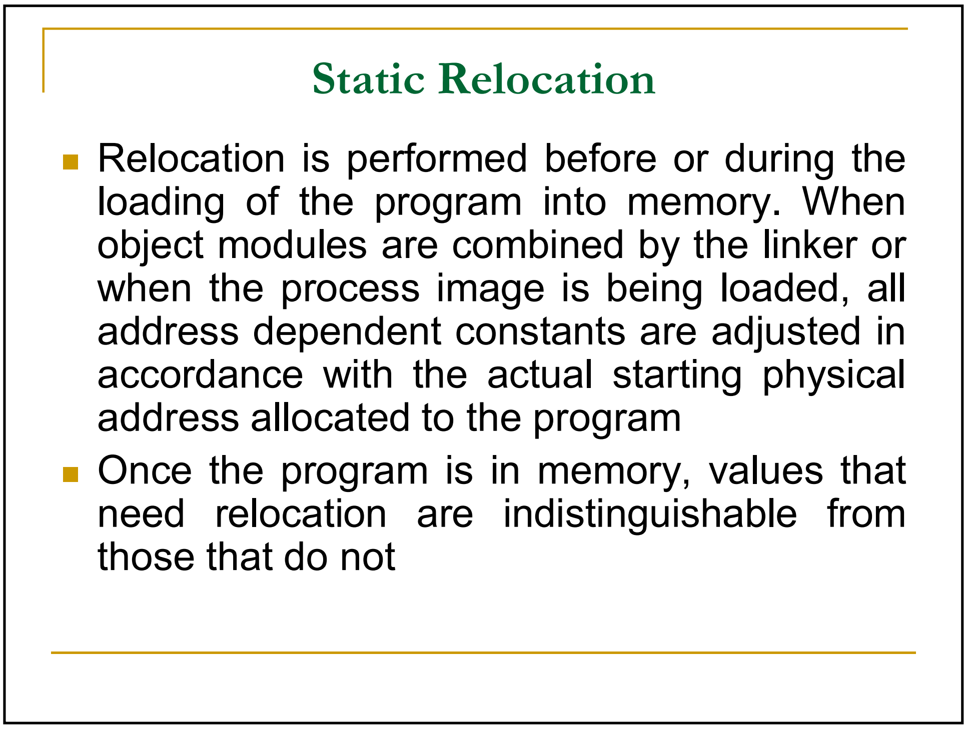
Swapping

- Regardless of the type of swapping file used, the need to access secondary storage makes swapping a lengthy operation relative to processor instruction execution
- OS that supports swapping usually cope up with this problem by allowing the process to be declared as swappable at the creation time ; however the authority to designate a process as un-swappable must be restricted to privileged users
- An important issue is whether the process to partition binding is static or dynamic



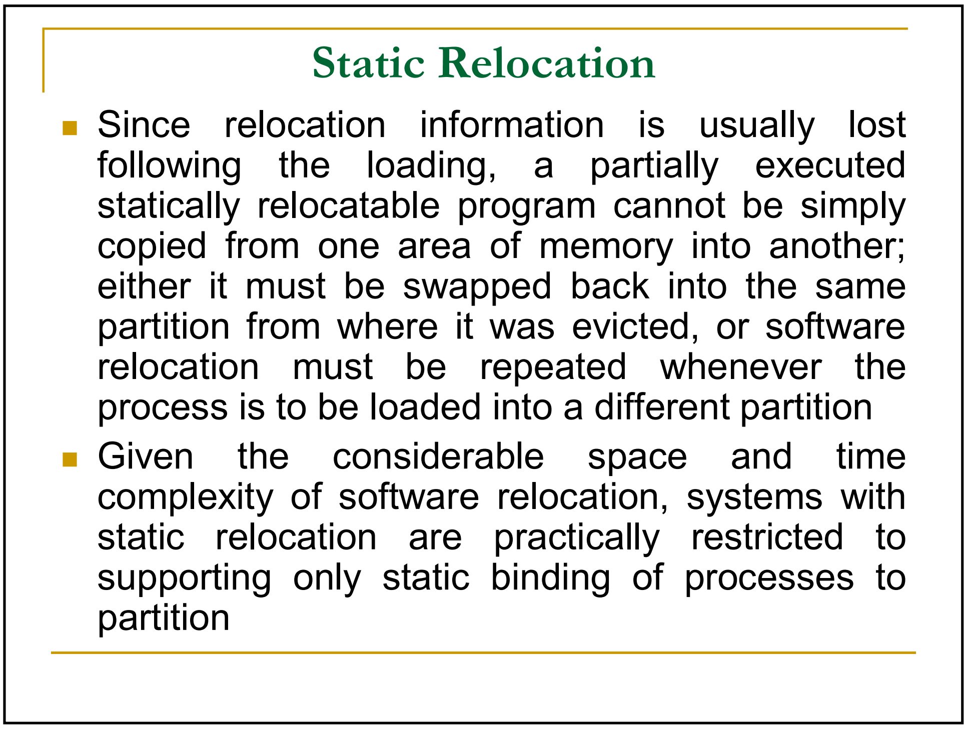
Relocation

- Program relocatability refers to the ability to load and execute a given program into an arbitrary place in memory as opposed to a fixed set of locations specified at program translation time
- The two basic types of relocation are :
  - Static
  - Dynamic



Static Relocation

- Relocation is performed before or during the loading of the program into memory. When object modules are combined by the linker or when the process image is being loaded, all address dependent constants are adjusted in accordance with the actual starting physical address allocated to the program
- Once the program is in memory, values that need relocation are indistinguishable from those that do not



Static Relocation

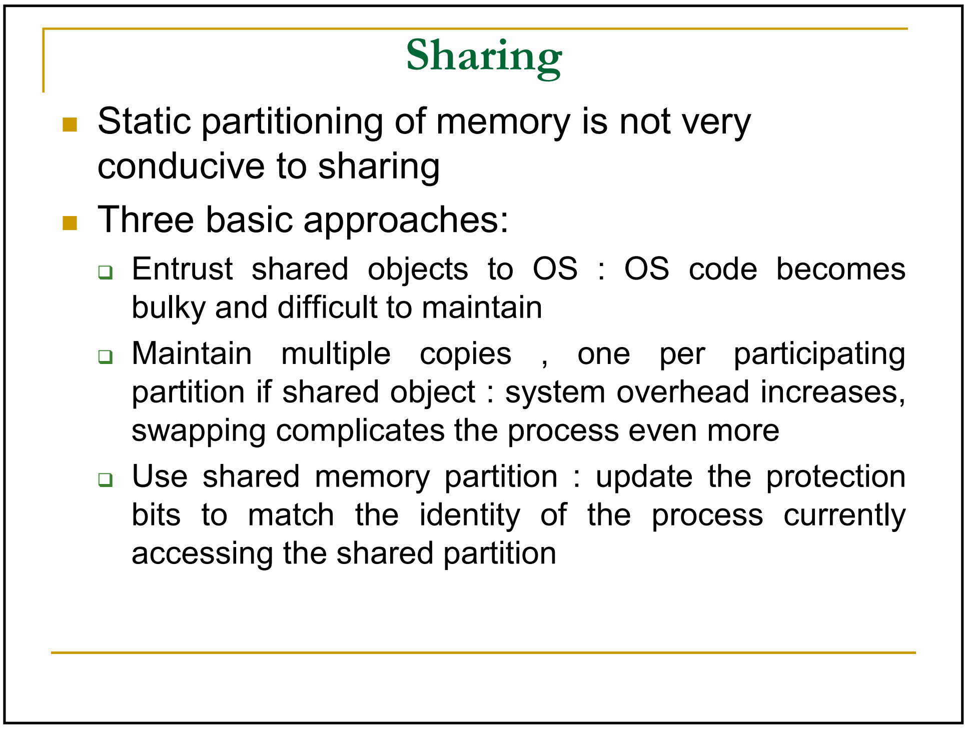
- Since relocation information is usually lost following the loading, a partially executed statically relocatable program cannot be simply copied from one area of memory into another; either it must be swapped back into the same partition from where it was evicted, or software relocation must be repeated whenever the process is to be loaded into a different partition
- Given the considerable space and time complexity of software relocation, systems with static relocation are practically restricted to supporting only static binding of processes to partition

Dynamic Relocation

- Mapping from the virtual address/relative address to the physical address space is performed at run time with some hardware assistance
- After allocating a suitable partition and loading a process image in memory, the OS sets a base register to the starting physical load address. Each memory reference generated by the executing process is mapped into the corresponding physical address by having the contents of the base register added to it
- Provides more flexibility to swapped out processes with the price of extra hardware and slowing down of the effective memory access time due to the relocation process

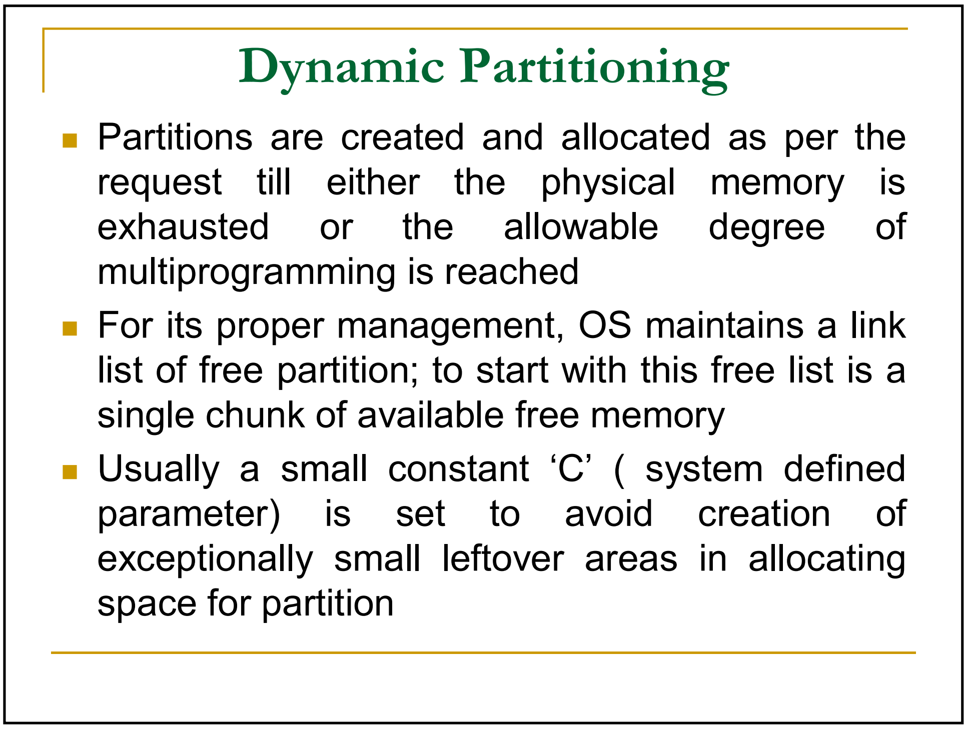
Protection

- The integrity of a system depends on its ability to enforce isolation of separate address space; system must be protected from unauthorized tampering by user processes, also each process must not inadvertently or maliciously access the areas of memory allocated to other processes
- Two common ways:
  - Base and limit register: base and limit values for each process kept in its corresponding PCB
  - Protection bits: process identity is recorded in the occupied blocks, system is assigned a unique key to allow unrestricted access to all blocks, no. of bits restrict no. of partition



Sharing

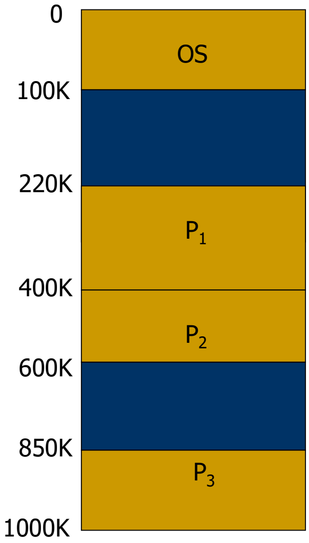
- Static partitioning of memory is not very conducive to sharing
- Three basic approaches:
  - Entrust shared objects to OS : OS code becomes bulky and difficult to maintain
  - Maintain multiple copies , one per participating partition if shared object : system overhead increases, swapping complicates the process even more
  - Use shared memory partition : update the protection bits to match the identity of the process currently accessing the shared partition



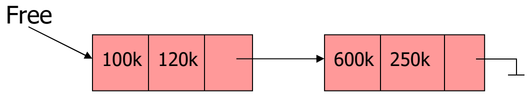
Dynamic Partitioning

- Partitions are created and allocated as per the request till either the physical memory is exhausted or the allowable degree of multiprogramming is reached
- For its proper management, OS maintains a link list of free partition; to start with this free list is a single chunk of available free memory
- Usually a small constant ‘C’ (system defined parameter) is set to avoid creation of exceptionally small leftover areas in allocating space for partition

Dynamic Partitioning



0	0 k	100 k	allocated
1	-	-	-
2	220 k	180 k	allocated
3	400 k	200 k	allocated
4	-	-	-
5	850 k	150 k	allocated



Dynamic Partitioning

- Common algorithms for selection of a free area of memory for creation of a partition are
  - First fit: Allocate the first partition/hole that is big enough
  - Best fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole
  - Worst fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
  - Next fit: A variant of Best fit; search begins from the last allocated free partition
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Dynamic Partitioning

- Usually when a free area is returned, it is recombined with adjacent free areas to make a single bigger chunk of free memory
- The patterns of returns of free areas is not same as the order of allocation. This leads to holes scattered all over memory in dynamic partitioning causing External fragmentation
- According to Knuth, approximately 33% or one-third of memory is wasted due to fragmentation
- Compaction is a solution but with added overhead

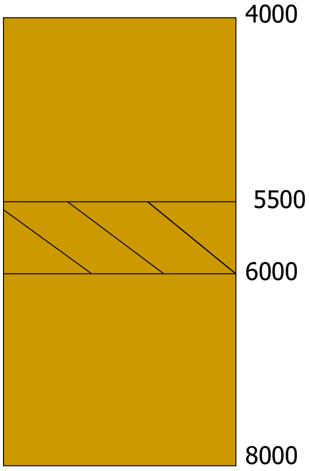
Dynamic Partitioning – Protection & sharing

- Almost same as static partitioning, except the difference that dynamic partitioning allows adjacent partitions in physical memory to overlap
- Such sharing is restricted to two processes only
- Shared code must ensure that references to itself are mapped properly during execution on behalf of any of the participating processes; code must either be position independent or occupy identical virtual offsets in address spaces of all processes that reference it

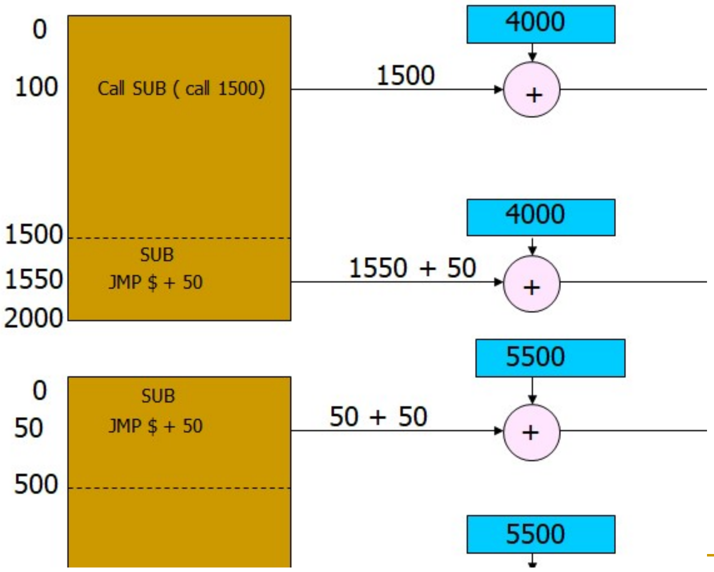
Dynamic Partitioning – Protection & Sharing

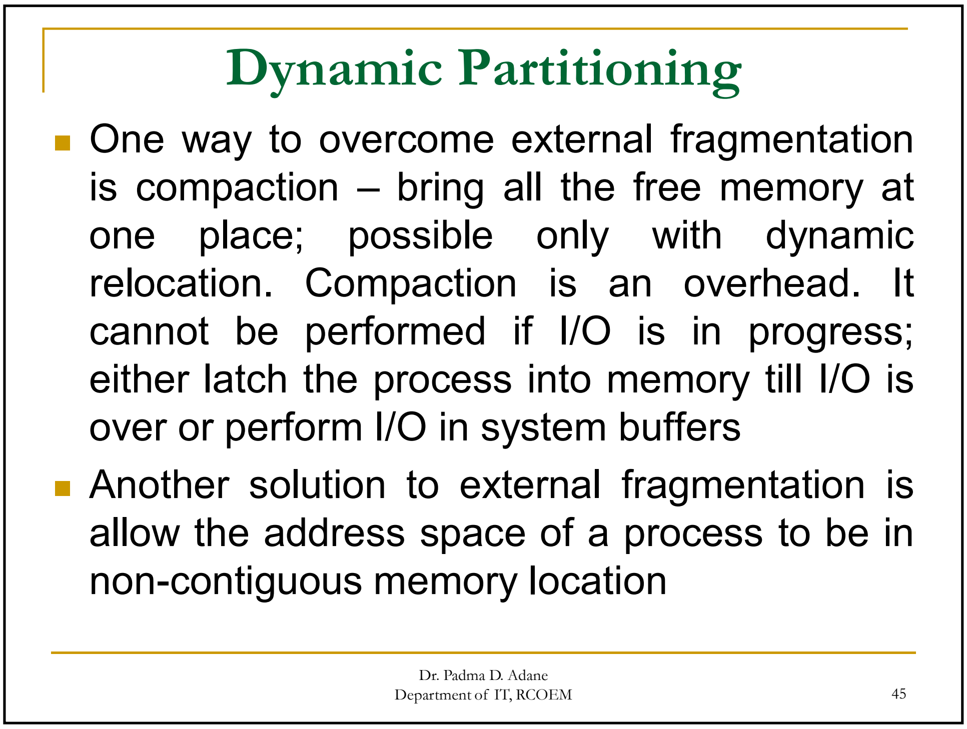
PDT

Base	size
4000	2000
5500	2500
- - - -	- - - -



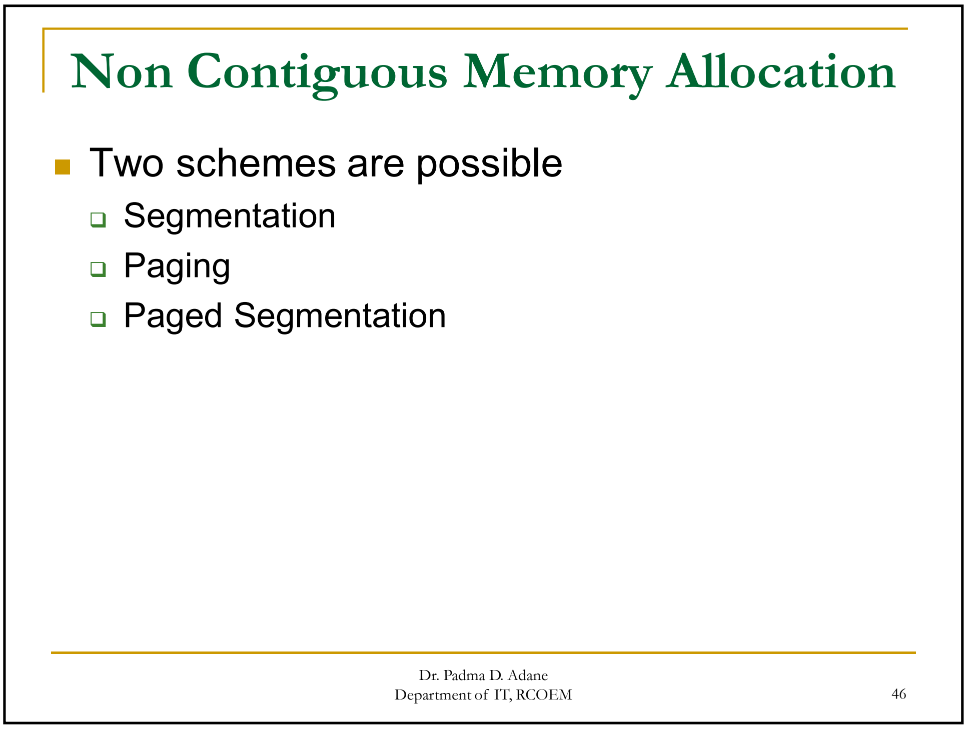
Dynamic Partitioning – Protection & Sharing





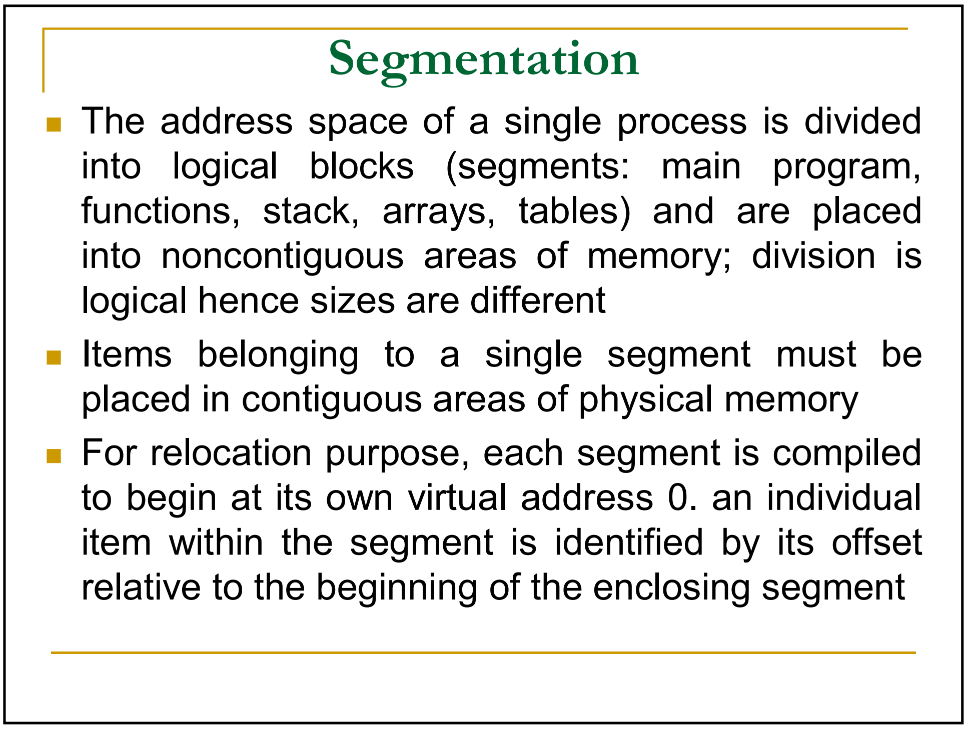
Dynamic Partitioning

- One way to overcome external fragmentation is compaction – bring all the free memory at one place; possible only with dynamic relocation. Compaction is an overhead. It cannot be performed if I/O is in progress; either latch the process into memory till I/O is over or perform I/O in system buffers
- Another solution to external fragmentation is allow the address space of a process to be in non-contiguous memory location



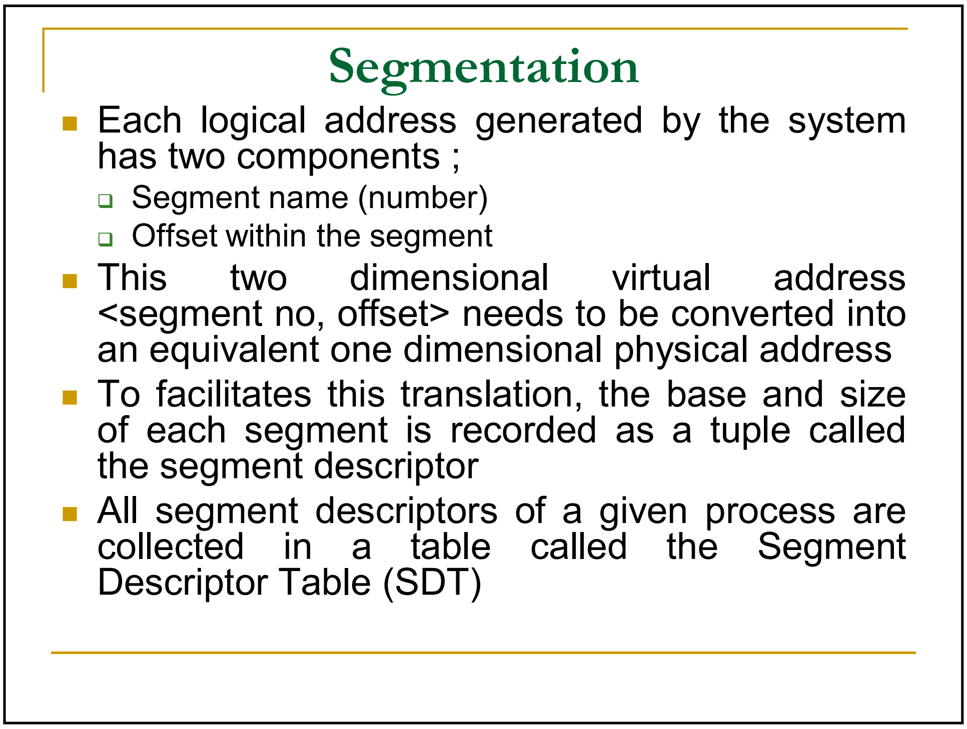
Non Contiguous Memory Allocation

- Two schemes are possible
  - Segmentation
  - Paging
  - Paged Segmentation



Segmentation

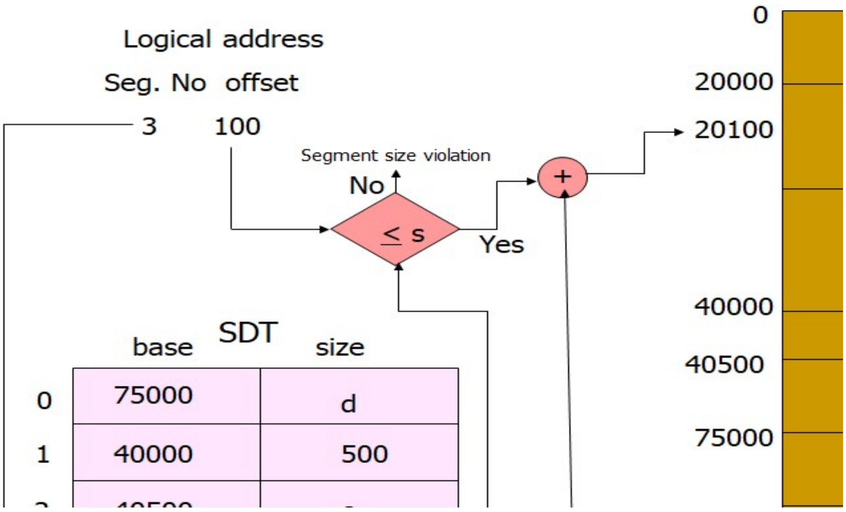
- The address space of a single process is divided into logical blocks (segments: main program, functions, stack, arrays, tables) and are placed into noncontiguous areas of memory; division is logical hence sizes are different
- Items belonging to a single segment must be placed in contiguous areas of physical memory
- For relocation purpose, each segment is compiled to begin at its own virtual address 0. an individual item within the segment is identified by its offset relative to the beginning of the enclosing segment



Segmentation

- Each logical address generated by the system has two components ;
  - Segment name (number)
  - Offset within the segment
- This two dimensional virtual address <segment no, offset> needs to be converted into an equivalent one dimensional physical address
- To facilitates this translation, the base and size of each segment is recorded as a tuple called the segment descriptor
- All segment descriptors of a given process are collected in a table called the Segment Descriptor Table (SDT)

Segmentation

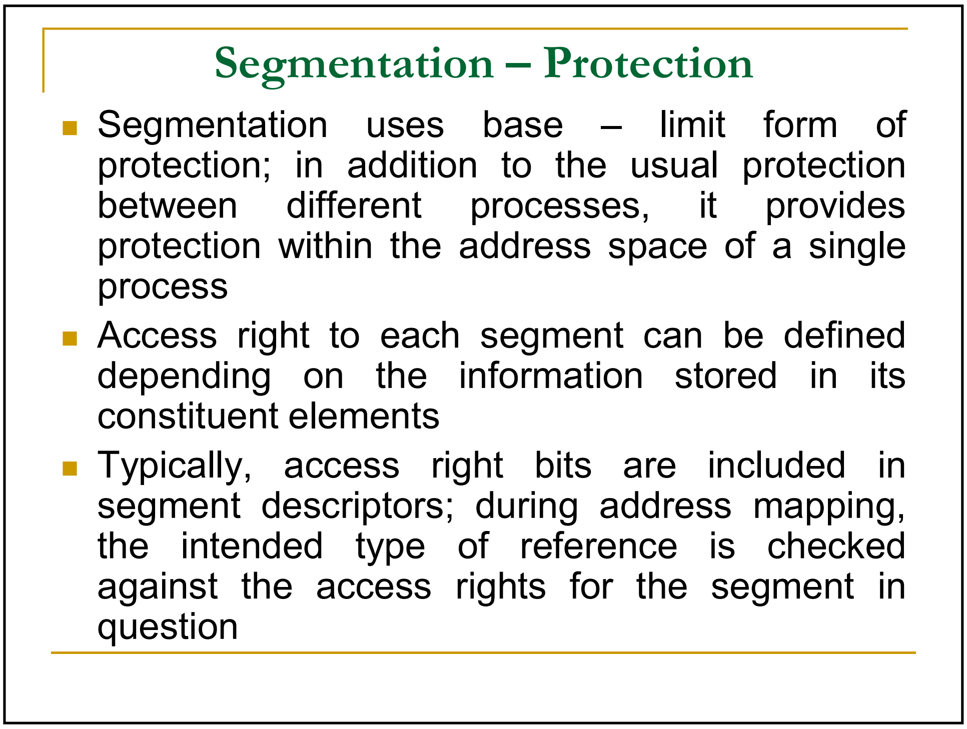


SDT

	base	size
0	75000	d
1	40000	500
2	40500	[illegible]

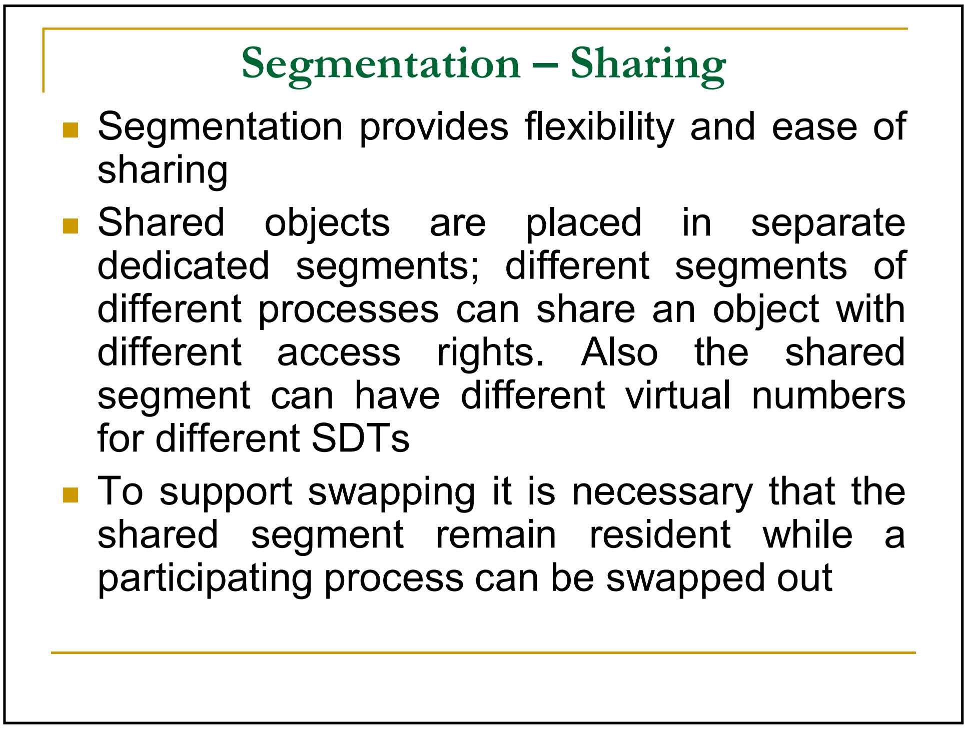
Segmentation

- The accessing of SDTs is facilitated by means of dedicated hardware registers called the SDTBR and SDTLR ; the base and limit value of SDT is kept in the PCB of the corresponding process
- Segmentation cuts memory bandwidth into half; mapping each virtual address requires two physical memory references
- To expedite translation, the frequently used segment descriptors can be kept in dedicated registers SDRs



Segmentation – Protection

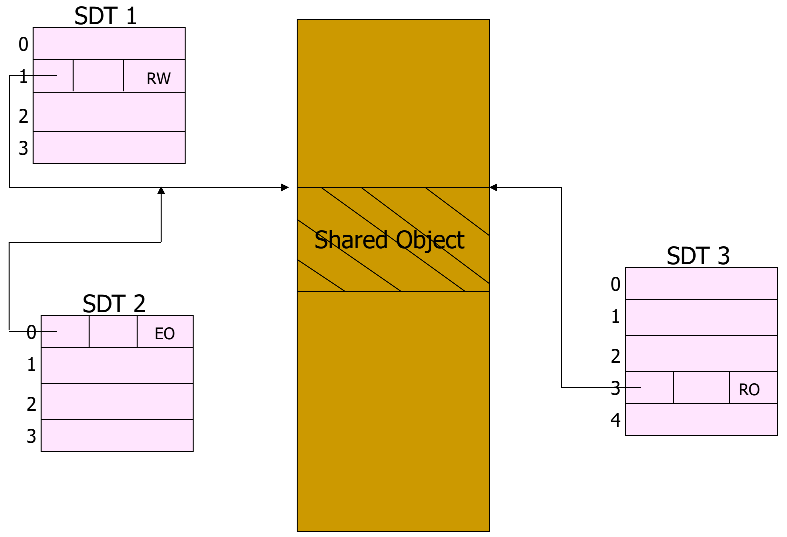
- Segmentation uses base – limit form of protection; in addition to the usual protection between different processes, it provides protection within the address space of a single process
- Access right to each segment can be defined depending on the information stored in its constituent elements
- Typically, access right bits are included in segment descriptors; during address mapping, the intended type of reference is checked against the access rights for the segment in question



Segmentation – Sharing

- Segmentation provides flexibility and ease of sharing
- Shared objects are placed in separate dedicated segments; different segments of different processes can share an object with different access rights. Also the shared segment can have different virtual numbers for different SDTs
- To support swapping it is necessary that the shared segment remain resident while a participating process can be swapped out

Segmentation – Sharing



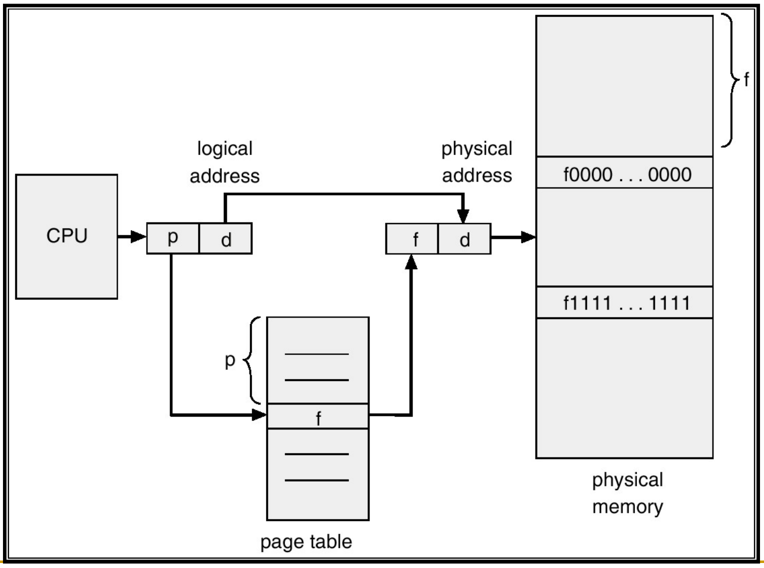
Paging

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes); Divide logical memory into blocks of same size called **pages**; the backing store(disk) is also divided into same size **blocks**
- To run a program of size *n* pages, need to find *n* free frames and load program; Keep track of all free frames
- Last page need not fit exactly in the frame; internal fragmentation
- Set up a page table to translate logical to physical addresses.

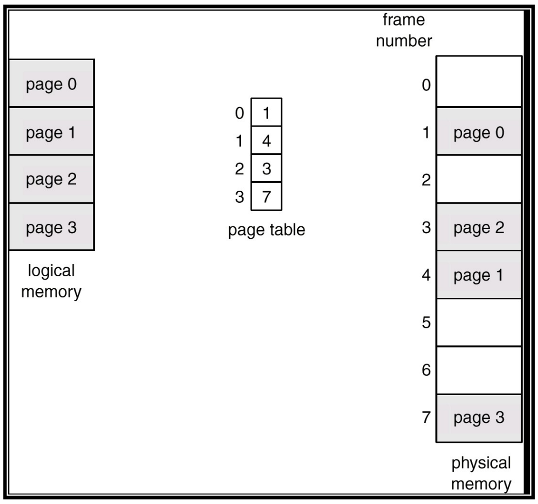
Paging

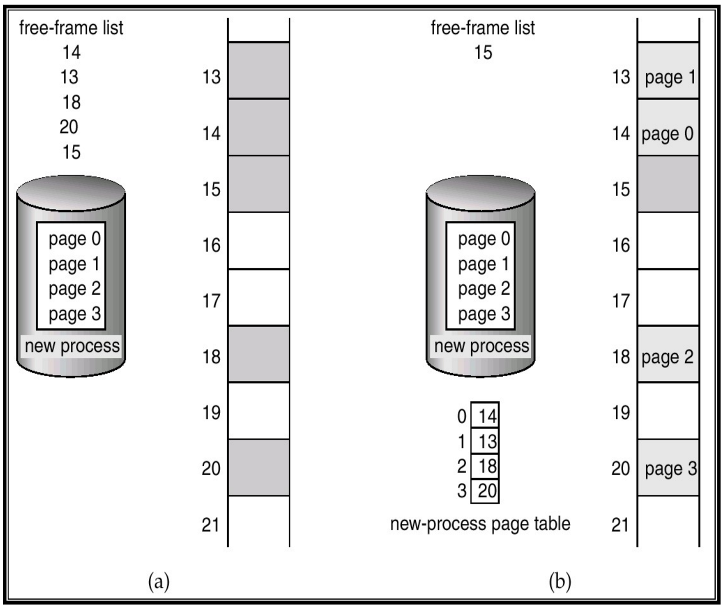
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.

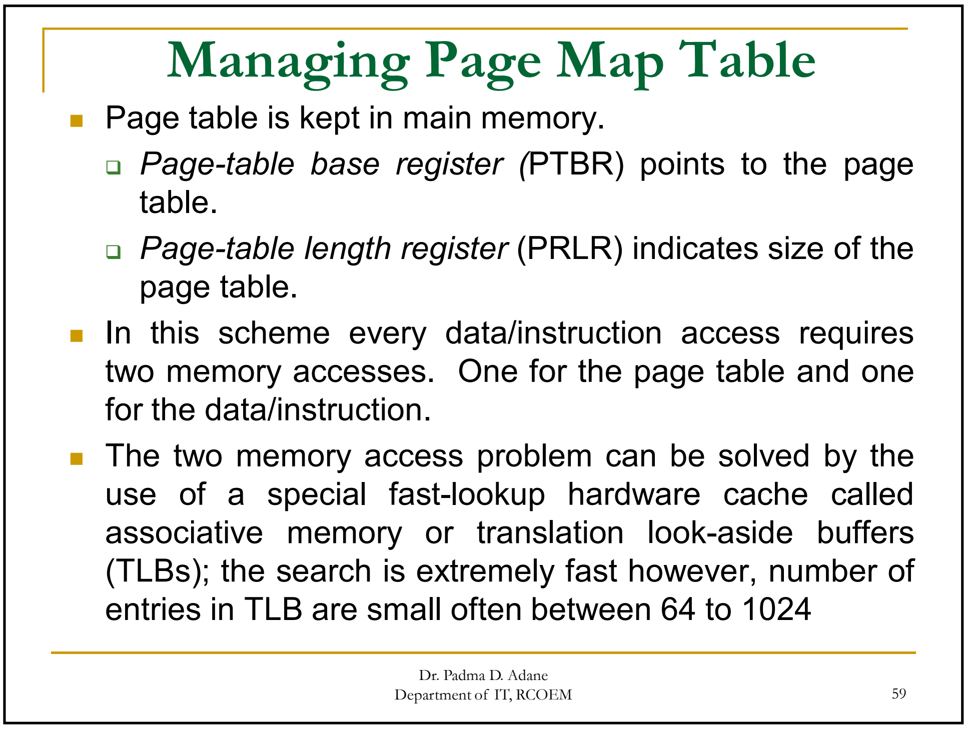
Paging



Paging

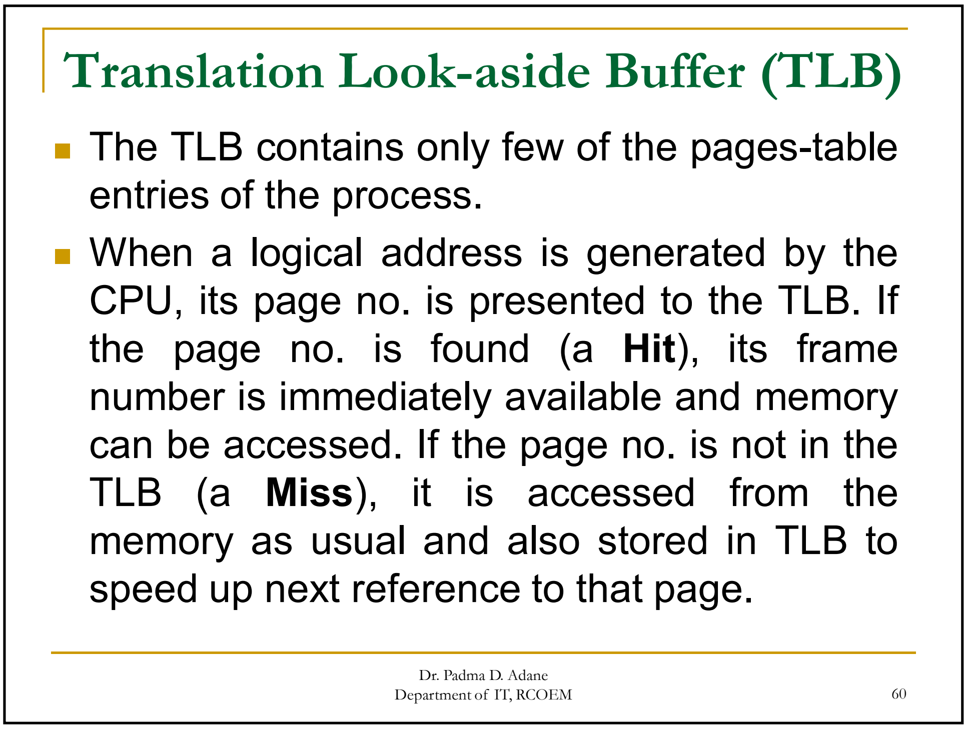

frame number
page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)



Managing Page Map Table

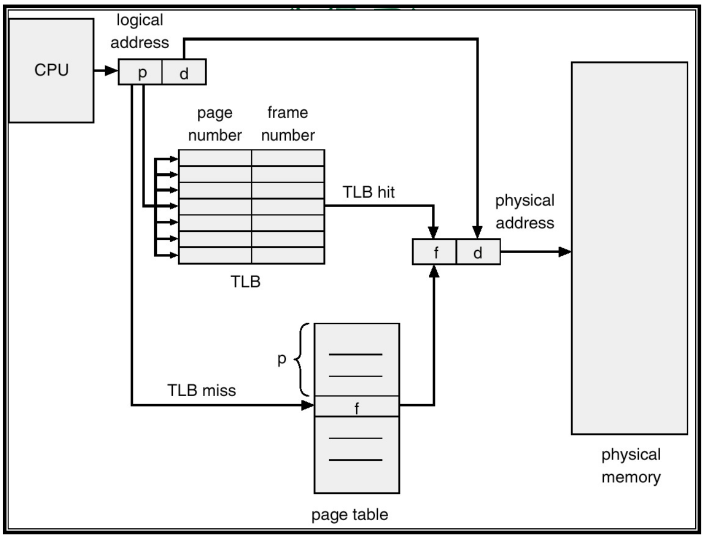
- Page table is kept in main memory.
  - *Page-table base register (*PTBR) points to the page table.
  - *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- 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); the search is extremely fast however, number of entries in TLB are small often between 64 to 1024



Translation Look-aside Buffer (TLB)

- The TLB contains only few of the pages-table entries of the process.
- When a logical address is generated by the CPU, its page no. is presented to the TLB. If the page no. is found (a **Hit**), its frame number is immediately available and memory can be accessed. If the page no. is not in the TLB (a **Miss**), it is accessed from the memory as usual and also stored in TLB to speed up next reference to that page.

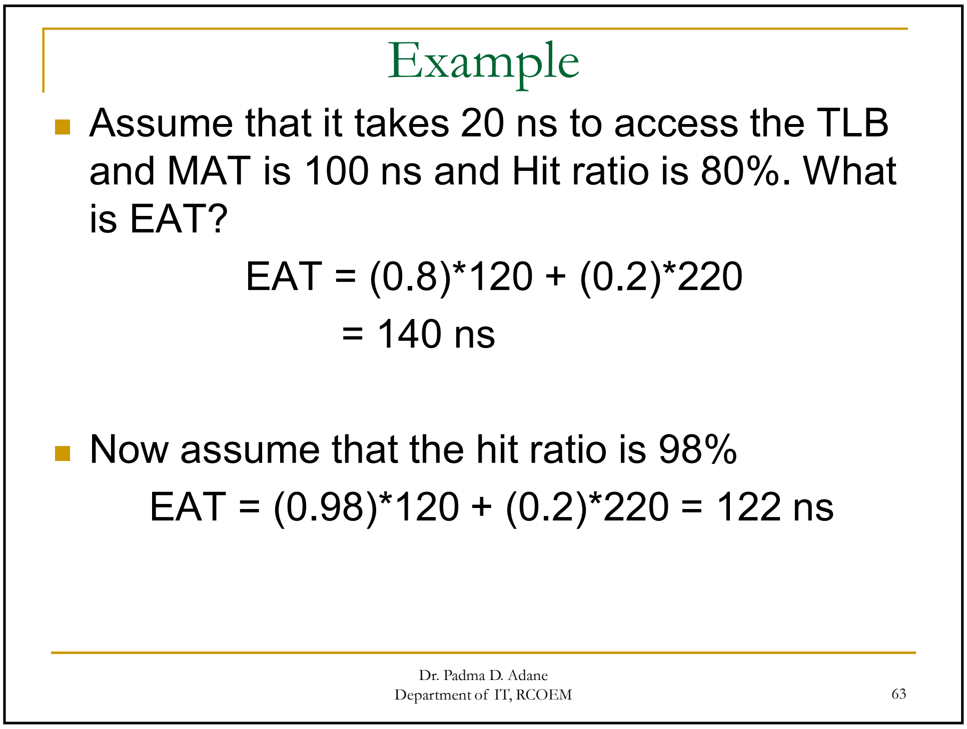
Translation Look-aside Buffer



Translation Look-aside Buffer (TLB)

- The percentage of times that a page number is found in the associative registers is called a Hit Ratio, it is related to the number of associative registers in the TLB.
- Assuming the Associative Lookup is ε time unit and Hit ration is α , The Effective Access time (EAT) is given as

$$EAT = (1\ MAT+ \varepsilon)\ \alpha + (2\ MAT+ \varepsilon)(1 - \alpha)$$



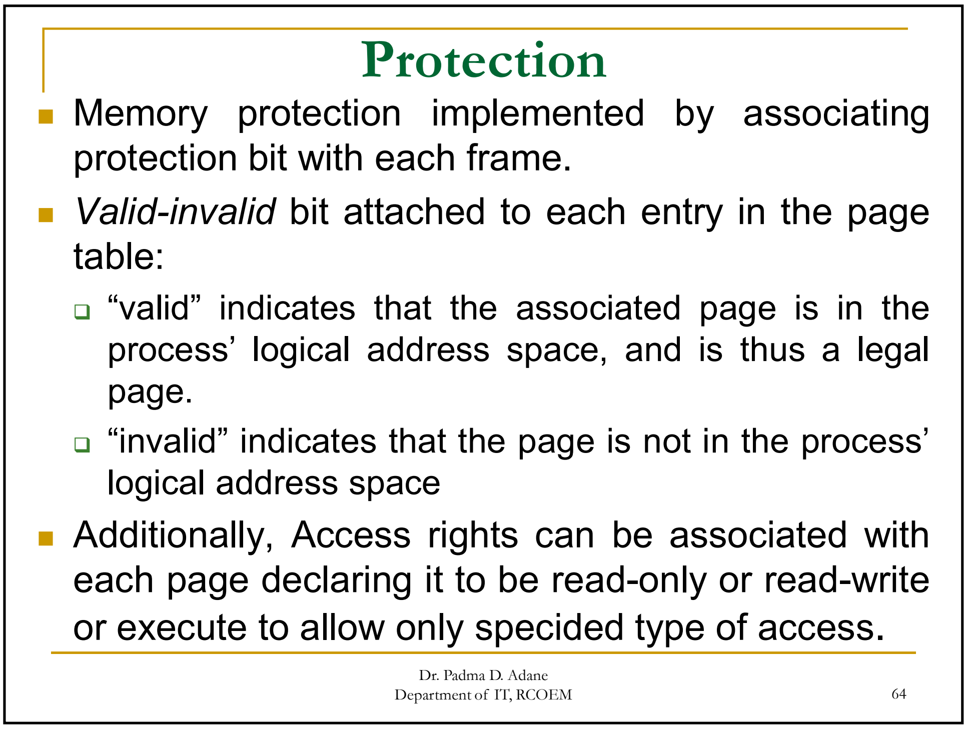
Example

- Assume that it takes 20 ns to access the TLB and MAT is 100 ns and Hit ratio is 80%. What is EAT?

$$EAT = (0.8)*120 + (0.2)*220$$
$$= 140 \text{ ns}$$

- Now assume that the hit ratio is 98%

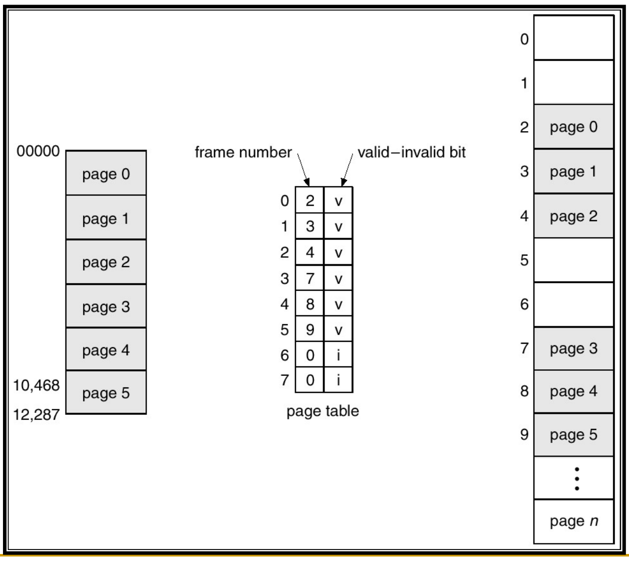
$$EAT = (0.98)*120 + (0.2)*220 = 122 \text{ ns}$$



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
- Additionally, Access rights can be associated with each page declaring it to be read-only or read-write or execute to allow only specided type of access.

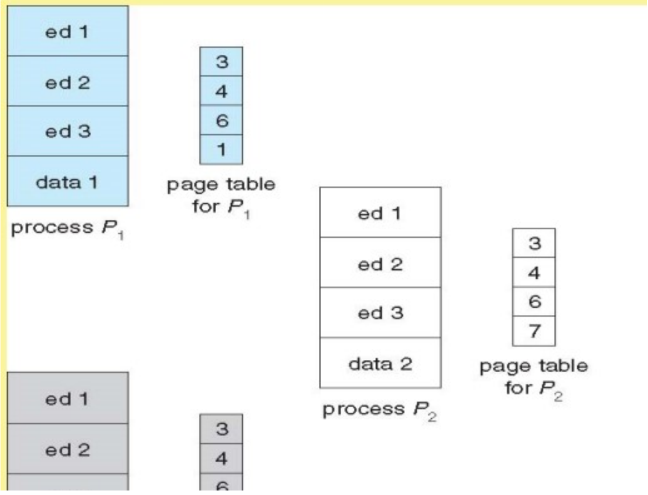
Valid (v) or Invalid (i) Bit In A Page Table



	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Sharing

- Sharing of read-only (reentrant) code is simple in paging



Structure of the Page Table

- In most modern computer systems, the logical address space is quite large. For example, consider a system with 32-bit (2^{32}), logical address space. If page size is 4 KB (2^{12}), then a page table may contain up to 1 million entries (2^{32} / 2^{12}). Assuming that each entry consists of 4 bytes, each process will need 4MB of space for its PMT.
- Contiguous allocation of space for PMT is not feasible.
- PMT itself can be in non-contiguous space.

Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

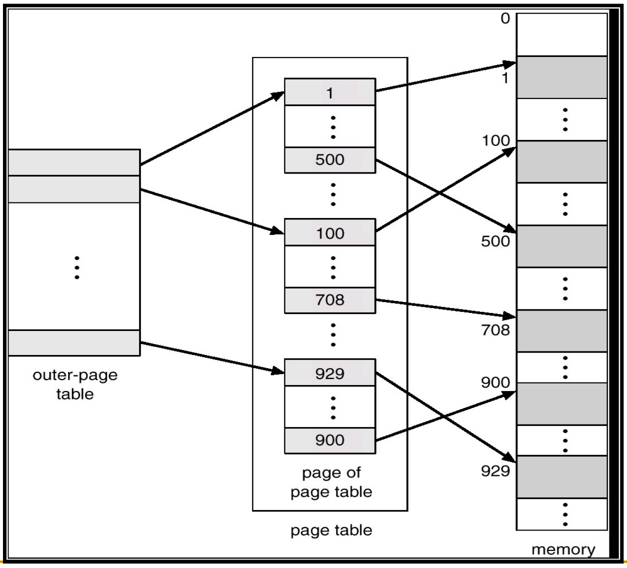
Hierarchical Paging

- Break up the logical address space into multiple page tables; A simple technique is a two-level page table.
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

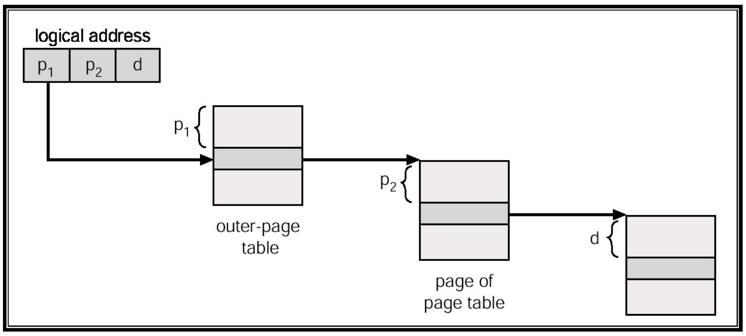
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme



Two-Level Page-Table Scheme

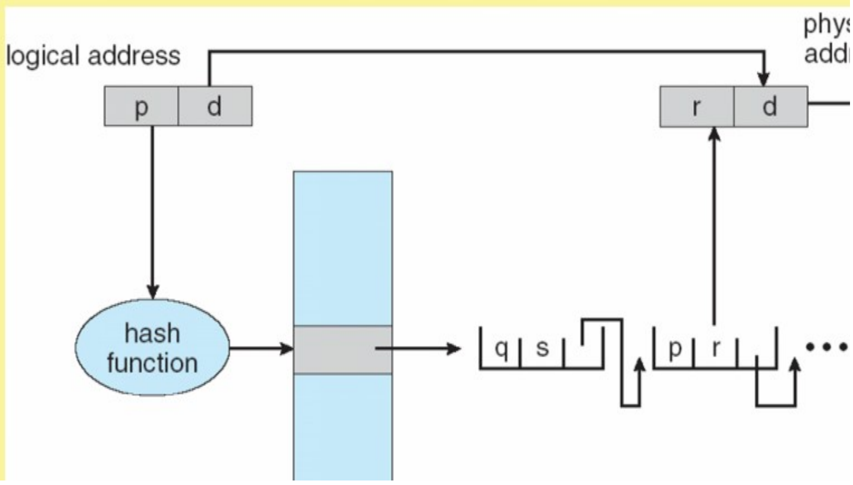
- Address-translation scheme for a two-level 32-bit paging architecture



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.

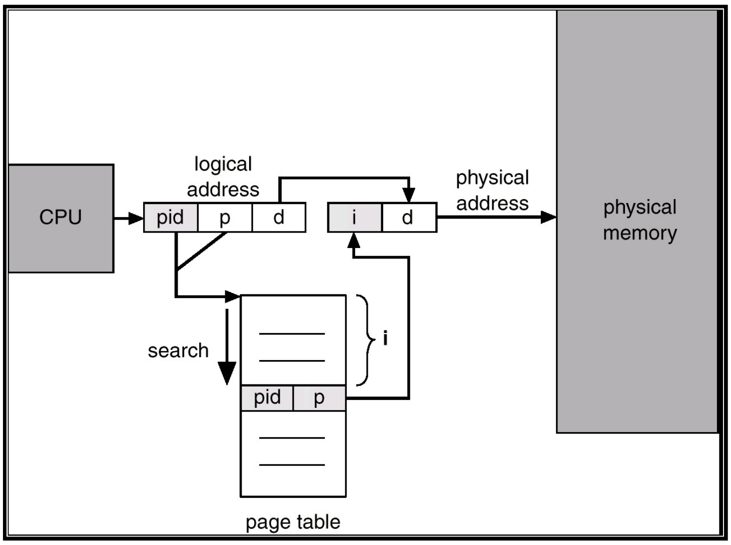
Hashed Page Tables



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; TLB can accelerate access

Inverted Page Table



Virtual Memory

Introduction

- Allows execution of partially loaded processes
- Virtual address space can be larger than the physical address space
- Frees the programmers from the concerns of memory-storage limitations
- Allows processes to easily share files and process creation
- Increases degree of multiprogramming and thus CPU utilization and throughput
- Not easy to implement
- Can be implemented with paging/segmentation as the underlying memory management schemes

Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Similar to paging with swapping
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

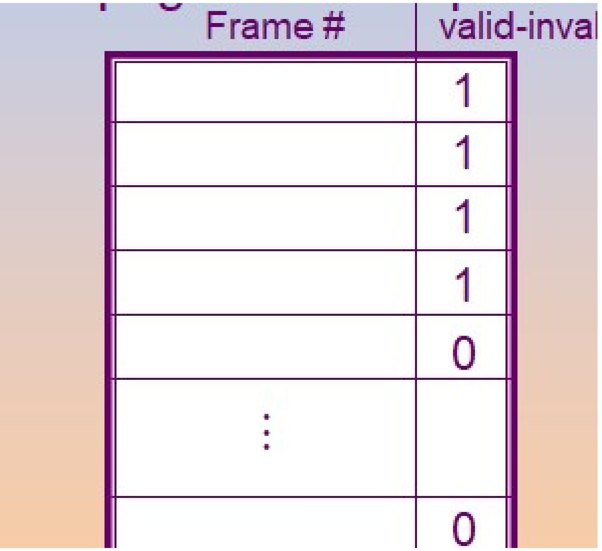
Demand Paging

- Lazy Swapper: Never swap a page into memory unless it is needed; a swapper that deals with pages is called a pager
- When a process is to be swapped in, the pager guesses which pages will be used; instead of swapping in a whole process, the pager brings in only those guessed pages into memory.
- Need new MMU functionality to implement demand paging so as to distinguish between pages that are in memory and pages that are on the disk.

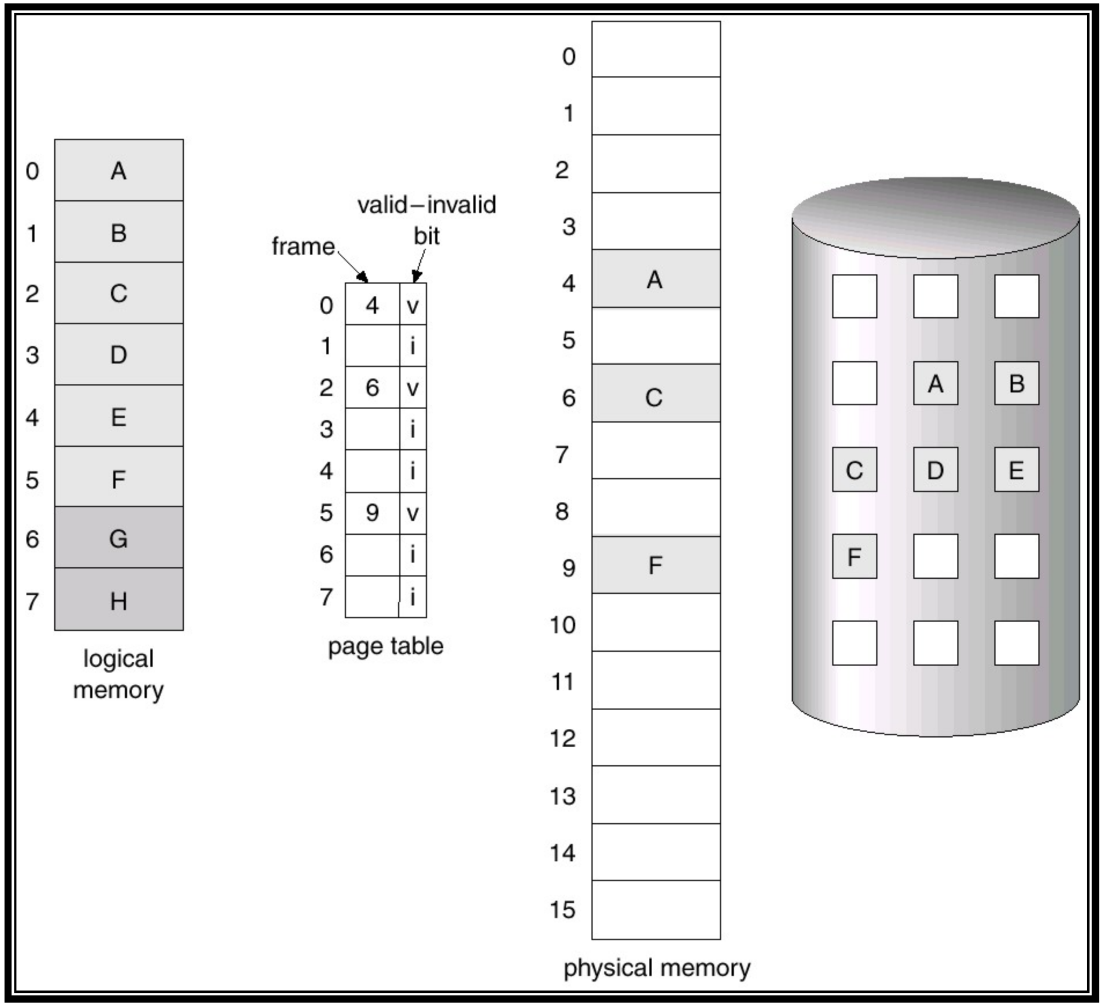
Demand Paging

- If pages are already memory resident, no difference in access from pure paging.
- If the page is not memory resident, it needs to be brought into the memory from secondary storage.
- With each page table entry a valid–invalid bit is associated; (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

Demand Paging



Frame #	valid-inval
	1
	1
	1
	1
	0
⋮	
	0



logical memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

page table

	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

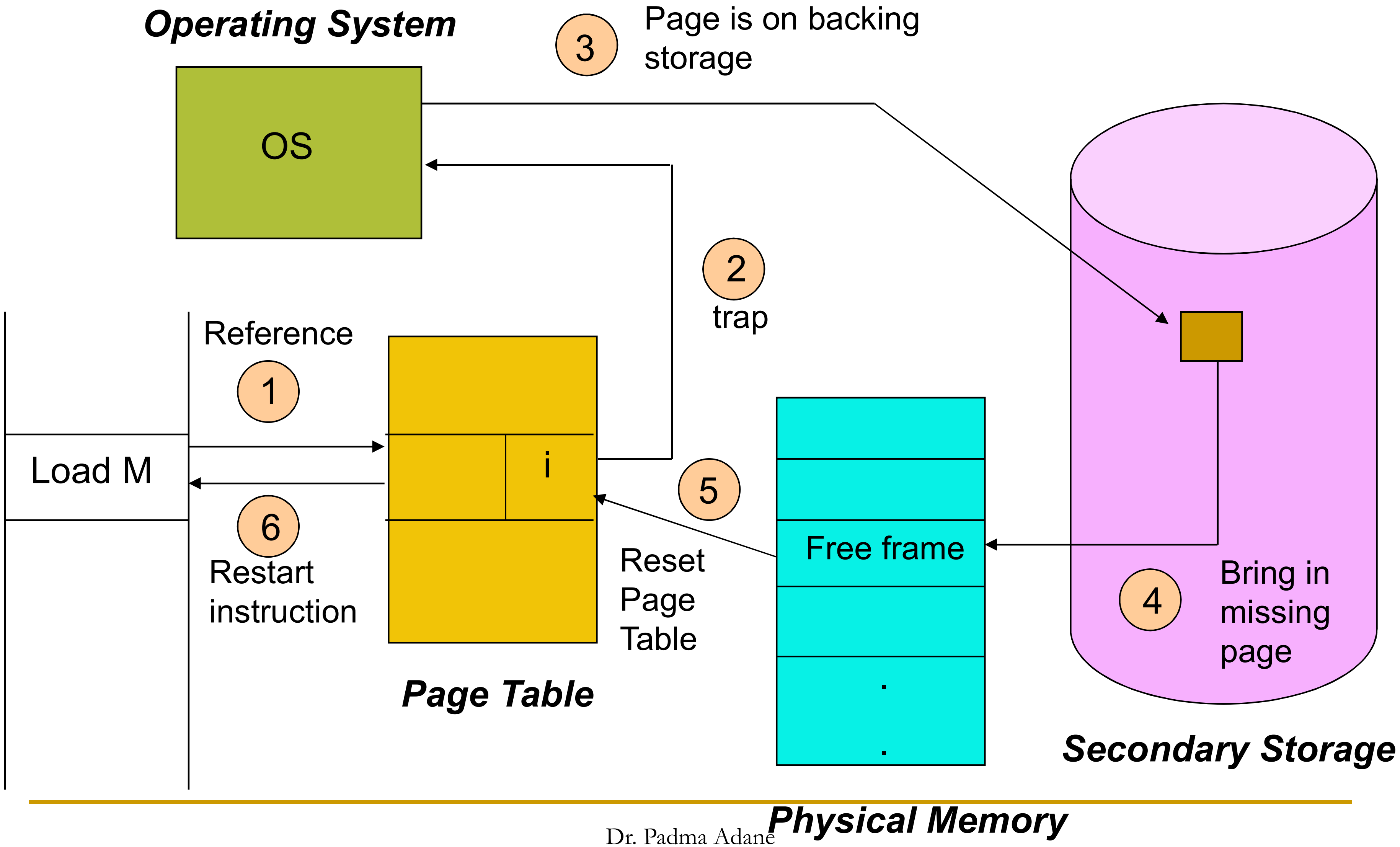
physical memory

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

Page Fault

- If there is ever a reference to a page, first reference will trap to
  OS ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Page not in memory ⇒ Get empty frame
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction

Steps in Handling a Page Fault



Instruction Interruptibility

Example 1

- Consider the execution of instruction

  C = A + B

- Converted to micro instruction :
  1. Fetch and decode the instruction (ADD)
  2. Fetch A
  3. Fetch B
  4. Add A and B
  5. Store the sum in C
- Page fault at instruction fetch can simple be restarted
- Page fault at operand fetch must fetch and decode the instruction again
- Page fault while storing the result at C must re-fetch and re-decode the instruction, undo the addition and add again

Instruction Interruptibility

Example 2

- Consider the machines using special addressing modes : autodecrement and autoincrement which automatically decrements and increments the contents before and after using the contents

  For e.g. MOV (R2)+, -(R3)

  if a page fault occurs while trying to store into the location pointed by R3, the values of the two registers must be reset to the values before the start of the execution of the instruction

Instruction Interruptibility

- At hardware level, instruction interruptibility can be handled in three ways :
  - Undo partial effects and restart the instruction; certain instruction cannot be restarted
  - Resume execution from exact point of interruption; requires involved storing
  - Pre-fetch memory references before starting the instruction; not justifiable for certain instructions
- Virtual memory technique can be implemented only in those machines that provide support for instruction interruptibility

Performance of Demand Paging

- Let *p* be the probability of page fault, *ma* be the memory access time, then the effective access time is given as :

  Effective access time = (1 – *p*) * *ma* + *p* * page fault time
- Page fault rate must be low to keep the effective memory access time low, else process execution slow downs dramatically
- Another important issue is disk I/O; disk I/O to swap space is generally faster than that to the file system; swap space is allocated in much larger blocks

Management of Virtual Memory

- Management of virtual memory requires incorporation of certain policies into the memory manager. These policies can be classified as:
  - **Allocation policy** : How much real memory to allocate to each active process
  - **Fetch policy** : Which items to bring and when to bring them from SS into MM
  - **Replacement policy**: Which page to evict in order to make room for the new one
  - **Placement policy** : where to place an incoming page

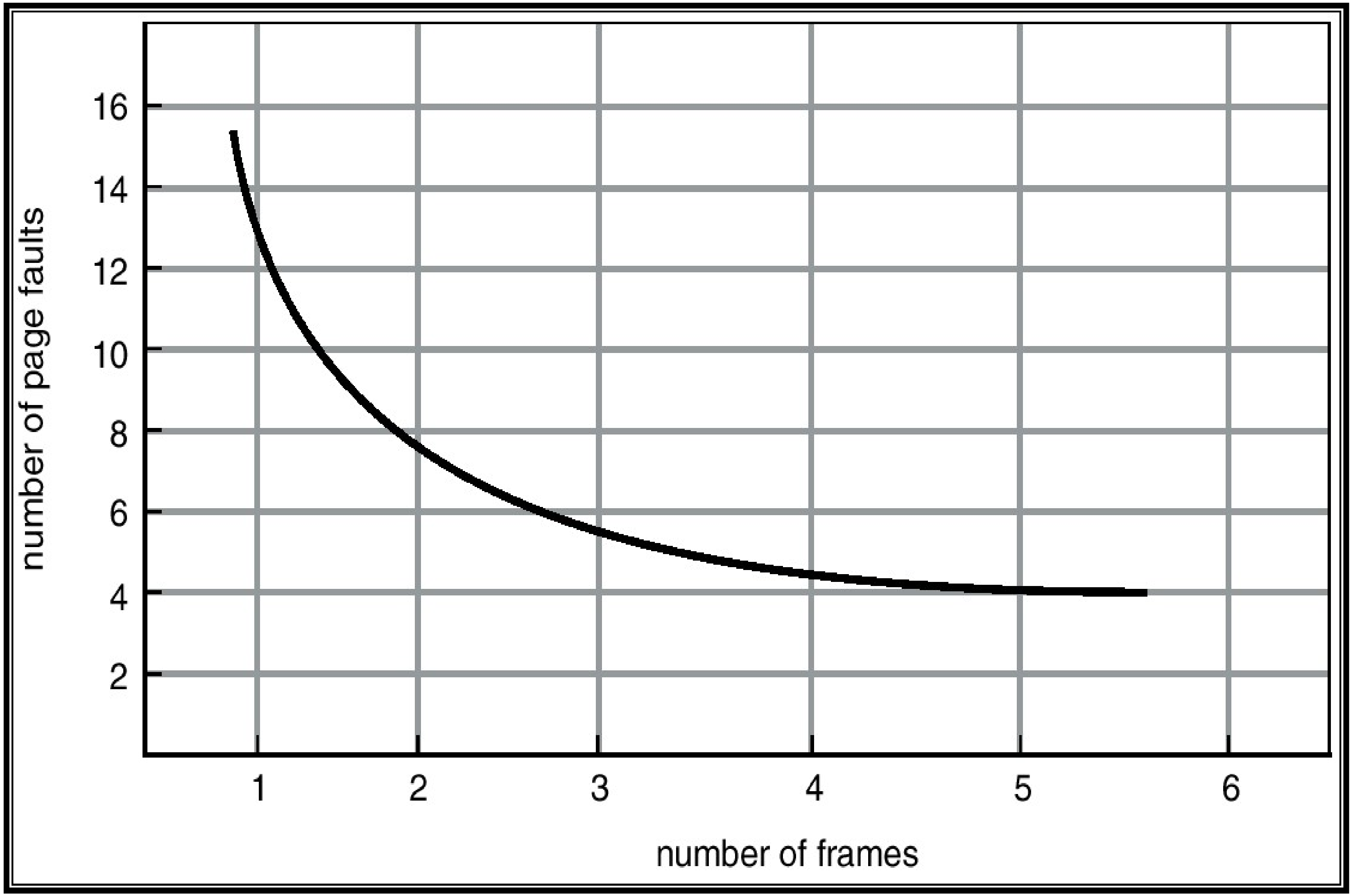
Management of Virtual Memory

- The performance of virtual memory systems is dependant in memory referencing patterns during program execution; missing item exceptions involve relatively long disk access delays increasing the turnaround times.
- Studies suggest a strong tendency of programs to favor subsets of their address space during execution : Locality of reference
  - Spatial Locality : tendency of a program to reference clustered locations in preference to randomly distributed locations
  - Temporal Locality : tendency of program to reference the same location or a cluster several times during brief intervals of time
- These findings are utilized for implementation of replacement and allocation policies.

Replacement Policies

- Two options to handle a missing page:
  - The faulted page may be suspended until some real memory becomes available
  - A page may be evicted to make room for the incoming one
- Missing item exceptions are fault of memory manger, not of the process itself. Suspending such processes even further adversely effects their scheduling and turnaround times. Hence former option is rarely chosen.
- Page Map table modified to include **Dirty Bit**.

Page Fault Vs Number of Frames



Replacement Algorithms First-In-First-Out

- The resident page spending longest time in memory is chosen when eviction is required
- Maintains a FIFO queue of pages for management
- Consider the following reference string

  144 A1 144 263 144 168 144 A1 179 A1 A2 263

FIFO Example

Reference String

144	A1	144	263	144	168	144	A1	179	A1	A2	263

Page Frames

PF0	144	144	144	144	144	168	168	168	179	179	179	179
PF1	-	A1	A1	A1	A1	A1	144	144	144	144	A2	A2
PF2	-	-	-	263	263	263	263	A1	A1	A1	A1	263

FIFO Queue

Front	144	A1	A1	263	263	168	144	A1	179	179	A2	263
	-	144	144	A1	A1	263	168	144	A1	A1	179	A2
Rear	-	-	-	144	144	A1	263	168	144	144	A1	179

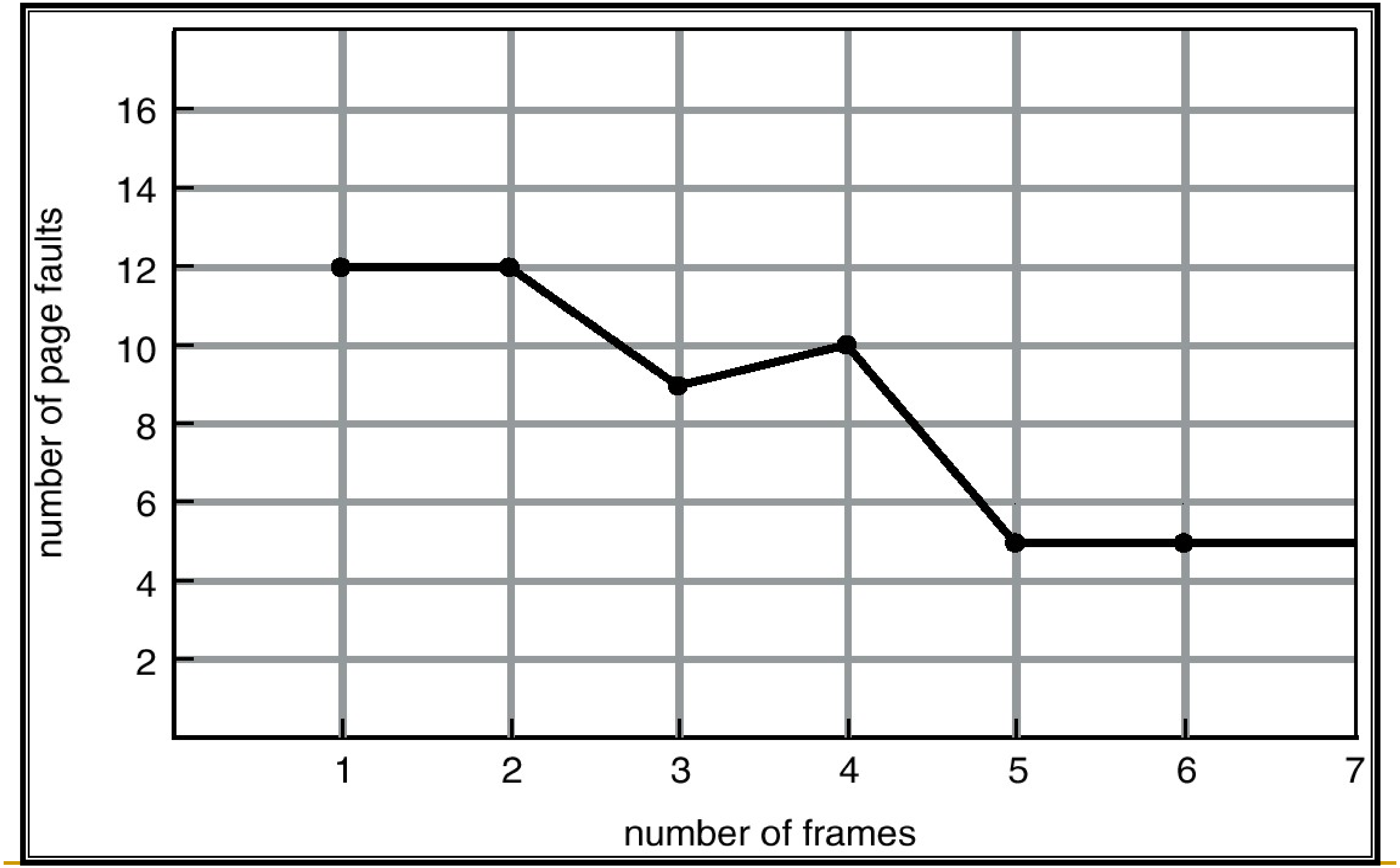
Number of Page Faults : 09

First-In-First-Out

- Consider the memory reference string:

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 3 frames no. of page fault is
  - 9
- With 4 frames the page fault increases to
  - 10
- Suffers from Belady's Anomaly : Number of page faults increases as the number of allocated frames are increased

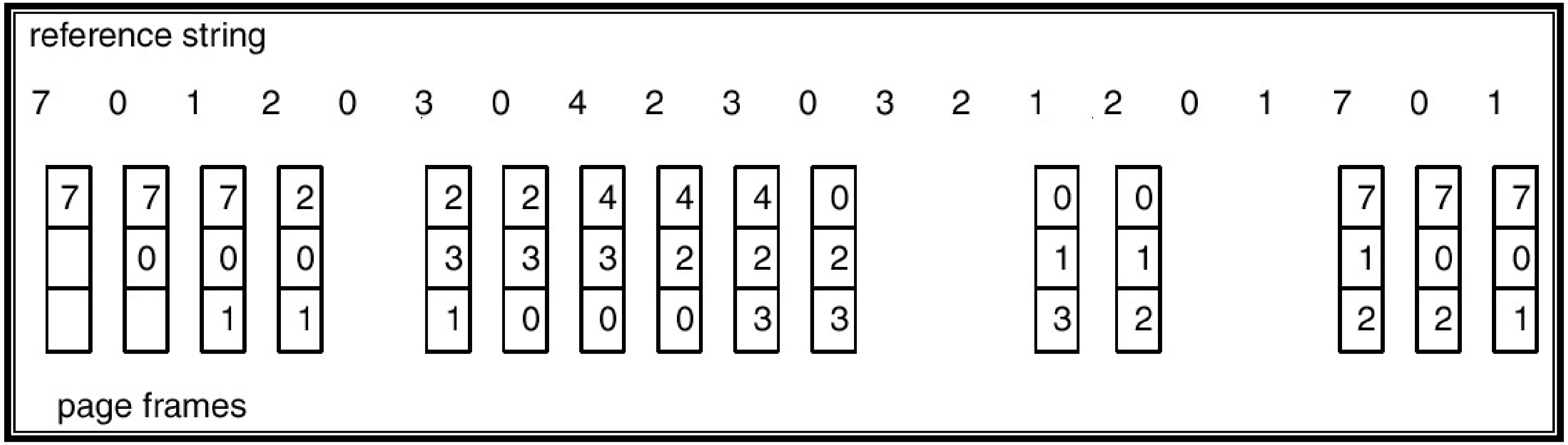
Belady’s Anomaly



FIFO Example

- Consider the reference string

  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 2 2 4 4 4 0 0 0 7 7 7

0 0 0 3 3 3 2 2 2 1 1 1 0 0

1 1 1 0 0 0 3 3 3 2 2 2 1

page frames

Number of Page fault is 15

Optimal

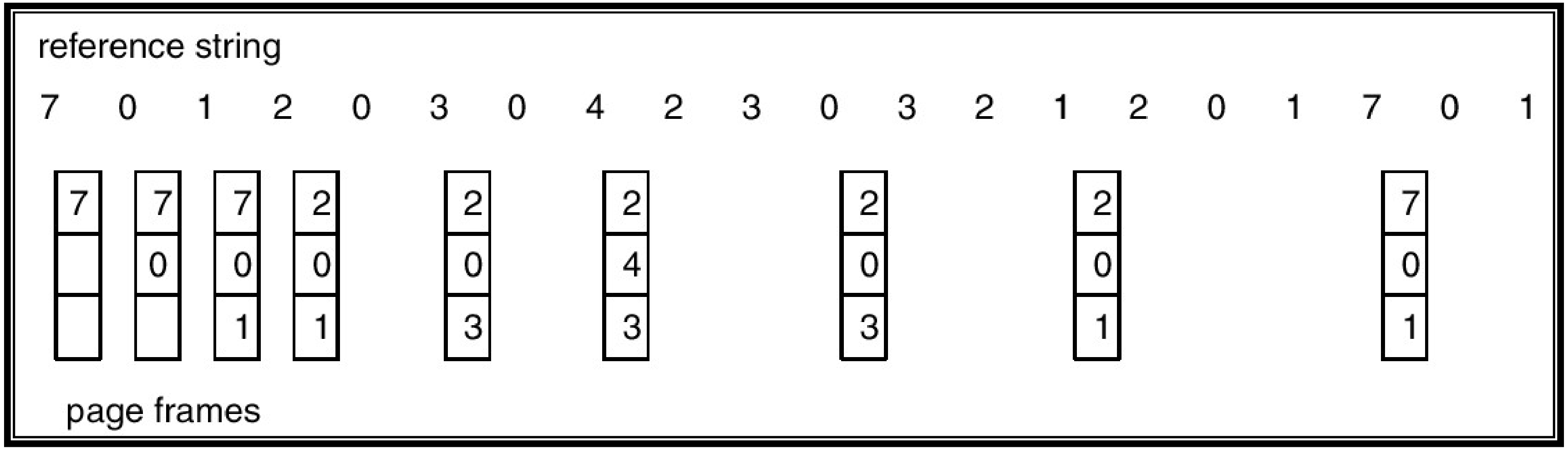
- Proposed by Belady, it removes the page to be referenced in the most distant future.
- Since the algorithm requires future knowledge, it is not realizable
- It has theoretical significance as it can serve as a yardstick for comparison with other algorithms

Example

- Consider the reference string

  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Let the no. of frames allotted be 3



Number of Page fault is 9

Least Recently Used

- Takes into account the patterns of program behavior and replaces the least recently used page
- On average performs better than FIFO
- Belongs to a large class of stack replacement algorithms; performs better when more memory is available; do not suffer from Belay’s anomaly

LRU Implementation

- Requires substantial hardware support
- Counter: with each PMT entry a time-of use field or counter is maintained. With each memory reference, the PMT is searched, the appropriate counter is incremented and written back.
- Stack: a stack of referenced page nos. is maintained, with each reference the corresponding page no. is put on the top, thus least recently used page is pushed at the bottom, implemented using doubly linked list

LRU Example

Reference String

144	A1	144	263	144	168	144	A1	179	A1	A2	263

Page Frames

PF0	144	144	144	144	144	144	144	144	144	144	A2	A2
PF1	-	A1	A1	A1	A1	168	168	168	179	179	179	263
PF2	-	-	-	263	263	263	263	A1	A1	A1	A1	A1

Reference Stack

	-	-	-	A1	A1	263	263	168	144	144	179	A1
	-	144	A1	144	263	144	168	144	A1	179	A1	A2
BOS	144	A1	144	263	144	168	144	A1	179	A1	A2	263

Number of Page Faults : 08

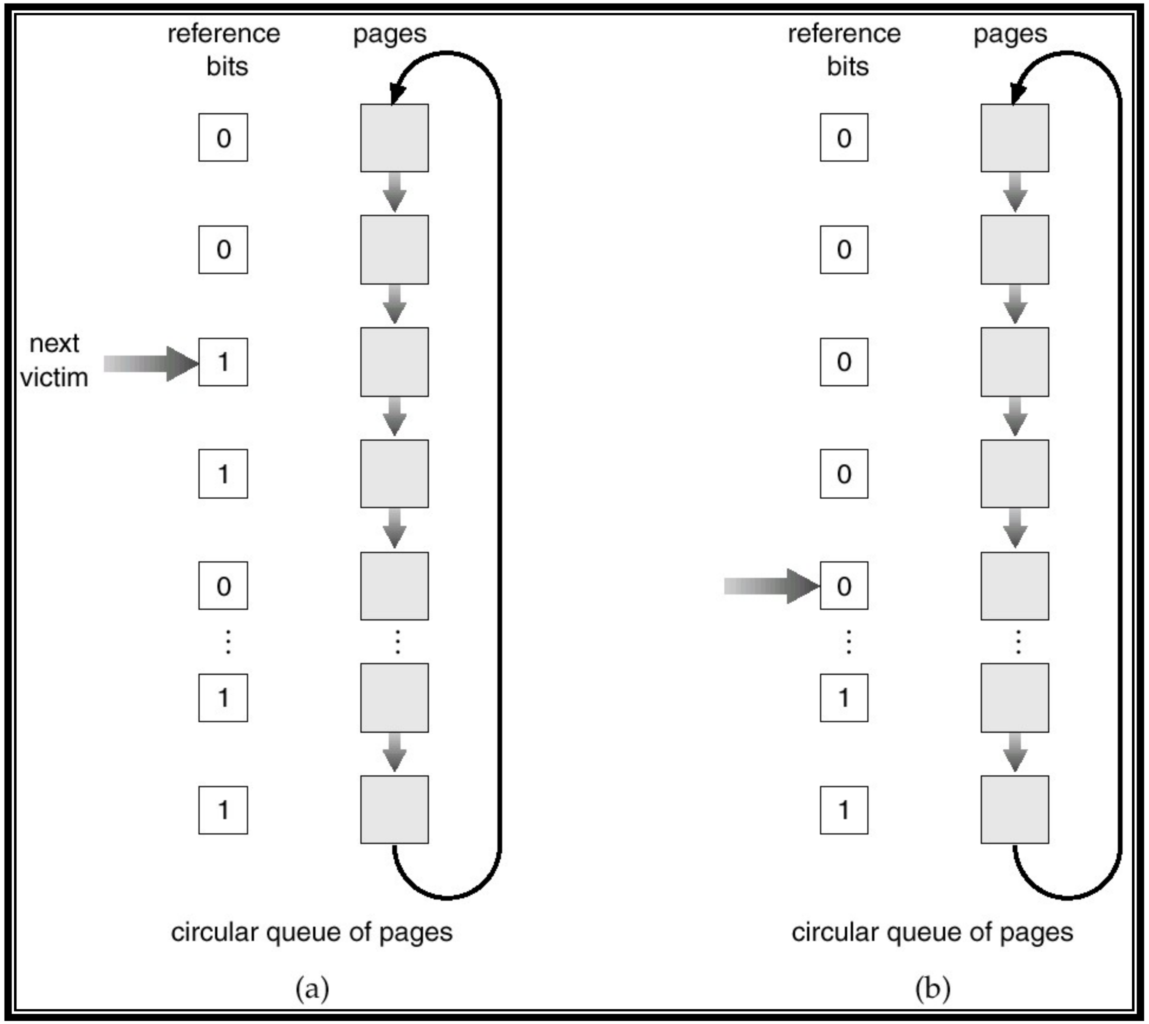
LRU Approximation Algorithms

- Reference Bit: Uses reference bit associated with each resident page.
- The bit is set whenever the related page is referenced and occasionally cleared by software.
- Setting a page indicates whether it has been referenced in the recent past; how recent this past is depends on the frequency of the referenced bit setting.
  - Initially the reference bit is 0 for all pages
  - When page is referenced, bit set to 1.
  - Replace the one which is 0 (if one exists).
    - We do not know the order, however.

LRU Approximation Algorithms

- Second Chance or Clock: A FIFO scheme with reference bit; can be implemented using a circular queue
- A pointer indicates which page is to be replaced next; when a page is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances it clears the reference bits.
- Once a victim is found, the page is replaced and the new page is inserted in that position.
- In the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance; degrades to FIFO replacement

LRU Approximation Algorithms



Enhanced Second Chance Algorithm

- An enhancement of clock algorithm, uses Reference bit as well as modified bit as an ordered pair
- With 2 bits per page, a page may fall in one of the four possible classes
  - (0,0) neither recently used nor modified; best choice
  - (0,1) not recently used but modified
  - (1,0) recently used but clean
  - (1,1) recently used and modified
- Logic same as clock algorithm, replaces the first page encountered in the lowest non-empty class; queue may be scanned several times before finding a page
- Scheme is used in Macintosh system

Counting based Page Replacement

- Keeps a count of the number of references made to each page
- Least frequently used : the page with smallest count is replaced; suffers from the situation where a page is used heavily during initial phase of the process and then never used again, the count will be large and page remains in memory although no longer needed
- Most frequently used : based on the argument that the page with smaller count was probably just brought in and is yet to be used

Page Buffering Algorithm

- System keeps a pool of free frames.
- When a fault occurs, a victim is chosen as before; however, the desired page is read into a free frame from the free pool before the victim is written out.
- Allows the process to restart at the earliest without waiting for the victim page to be written out.
- When the victim is later written out, its frame is added to the free-frame pool

Page Buffering Algorithm

- An expansion of this scheme is to maintain a list of modified pages.
- Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modified bit is then reset.
- This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Global Vs Local Replacement

Global	Local
When selecting a victim, processes all resident pages regardless of the owner's identity	A victim is selected only from the processes own set of allocated frames
Increases the degree of coupling between replacement and allocation strategies. Pages allocated to one process by allocation algorithm may be taken away by a global replacement algorithm. Thus may affect the premises on which the logic and properties of the replacement algorithm in question are based	Localizes the effects of the allocation policy to each process

Global Vs Local Replacement

Global	Local
The set of pages in memory for a process not only depends on the its own paging behavior but also on the behavior of other processes. Thus a process cannot control its own page fault.	The set of pages in memory for a process is effected by the paging behavior of that process only. It hides a process by not making its less used pages in memory available to other.

Global replacement generally results in greater throughput and is therefore the more common method.

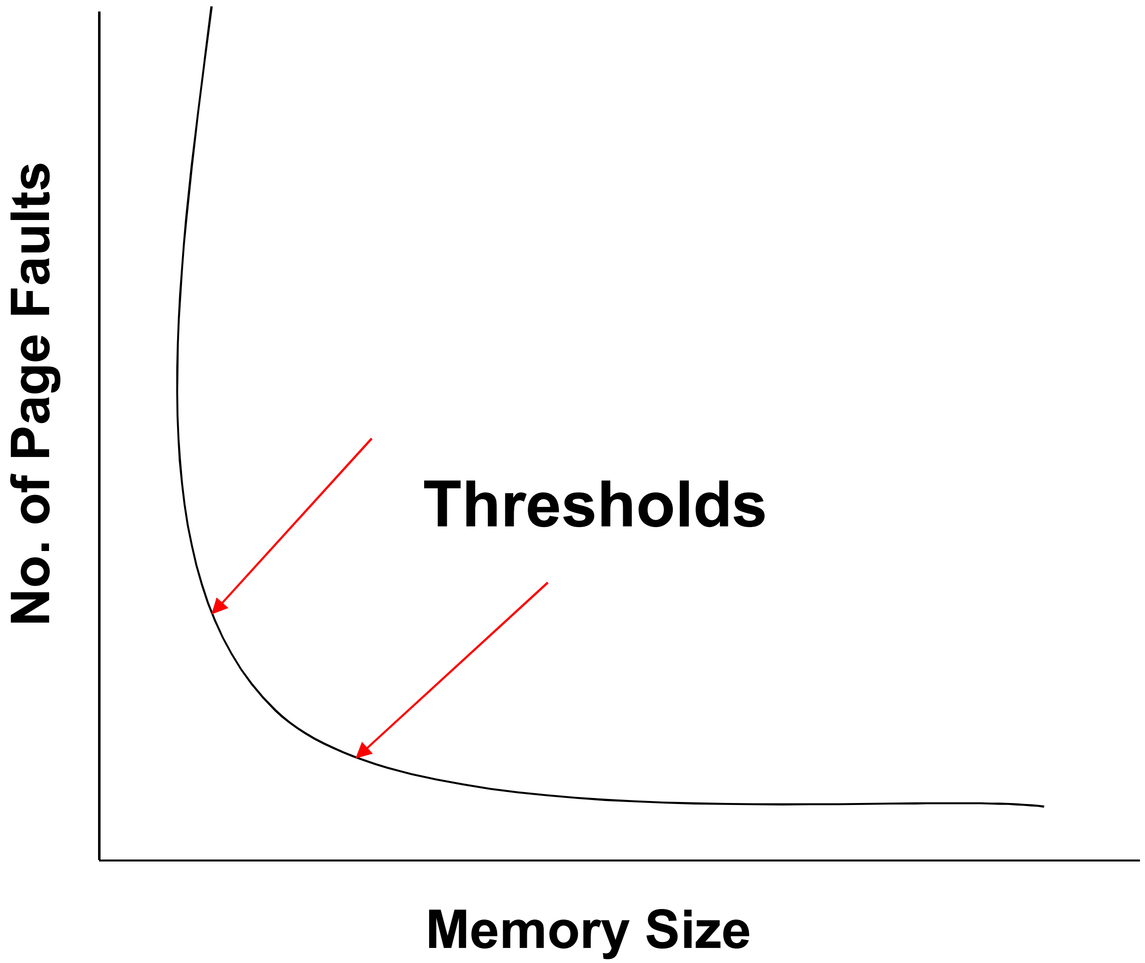
Allocation Policies

- Governs the OS decisions regarding the amount of real memory to be allocated to each active process.
- Must compromise among several conflicting requirements.
- In terms of paging, giving more real pages to a process should result in reduced page fault frequency and improved turnaround times. However, generous allocation of pages may effect degree of multiprogramming lowering the processor utilization.
- Too few pages allocated to a process may increase page fault rates and deteriorates the turnaround times to unacceptable levels.

Allocation Policies

- The lower limit on the number of pages allocated is architecture dependent; fewer pages allocated than the maximum size of an instruction may cause the instruction to fault continuously .
- The relation ship between the frequency of page faults and the amount of real memory allocated to a program in nonlinear.
- Each program has a certain threshold below which the no. of page fault increases very quickly and above which allocation of additional memory results in moderate to little performance improvement
- Allocation of memory should be such that each active program is between these two thresholds.

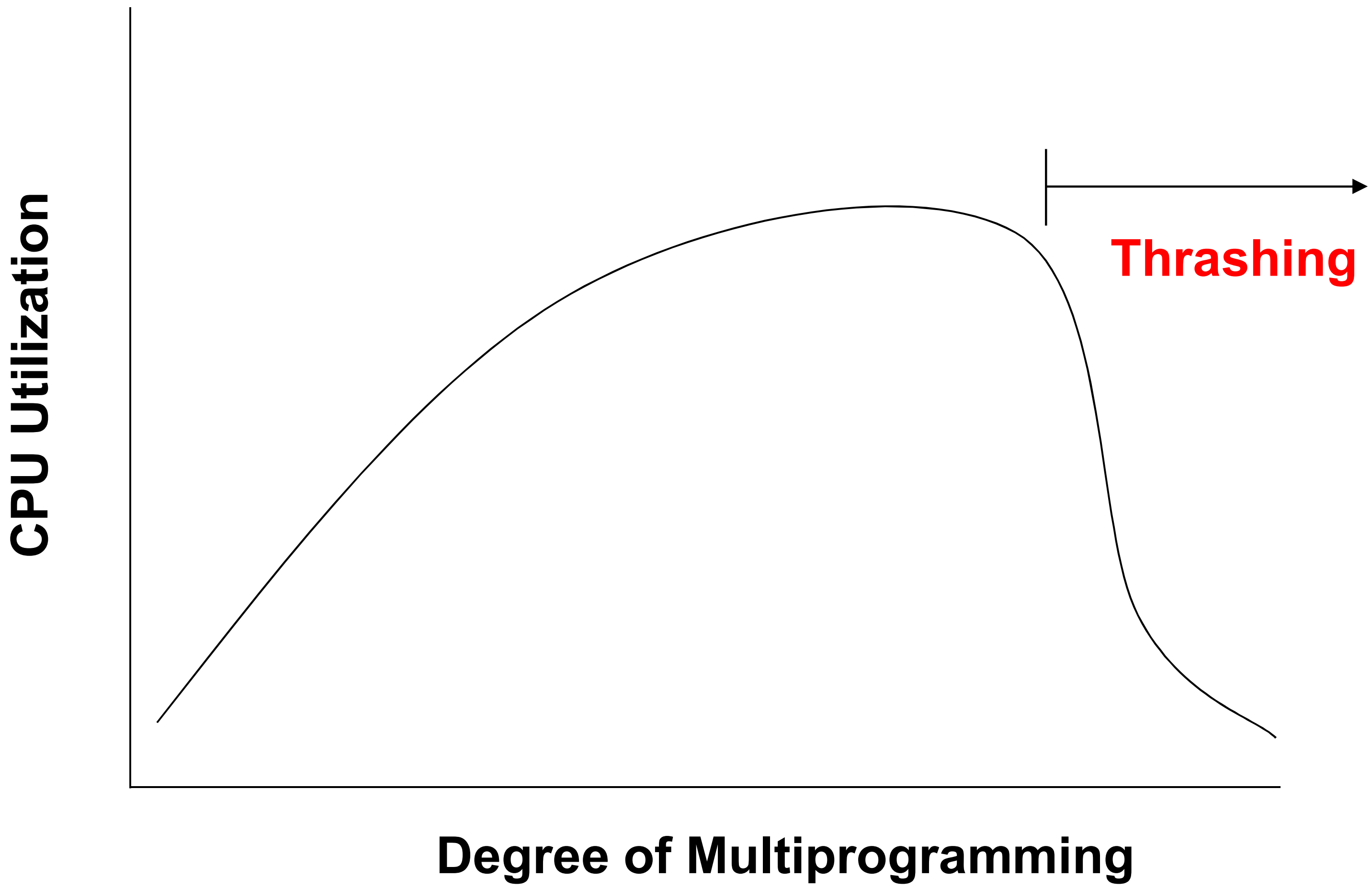
Allocation Policies
Parachor Curve



Thrashing

- A high paging activity caused by over allocation of memory frames; a process in thrashing spends more time paging than executing.
- After observing a low processor utilization, the OS may attempt to improve it by activating more processes. If no free page frames are available, the holdings of the already active processes may be reduced (global replacement). This may drive some of the processes into high page fault zone, lowering the processor utilization further. In order to improve the still decreasing processor utilization, the OS may decide to further increase the degree of multiprogramming :The system is heading towards Thrashing

Thrashing



Thrashing

- At the point of thrashing, OS must decrease the degree of multiprogramming.
- The effects of thrashing can be limited by using local replacement algorithms. Still, a thrashing process holds paging device too long affecting the access time of processes that are not thrashing.
- A good design must ensure that the allocation algorithms are not inherently unstable and inclined towards thrashing; too few pages may lead to thrashing and too many pages may unduly restrict the degree of multiprogramming and processor utilization

Allocation policies

- Two allocation policies which address the problem of Thrashing are :
  - Page Fault Frequency (PPF)
  - Working Set Theory

Page Fault Frequency

- Define the upper and lower page fault rates for each process; the actual page fault rates may be measured and recorded in the PCB of the process.
- When a process exceeds the upper page fault frequency threshold , more real pages may be allocated to it; as soon as the process responds by reaching the lower PFF threshold, allocation of new page frames to it may be stopped.
- The PPF parameter P may be defined as **P= 1 / t** where T is the critical inter page fault time. P is usually measured in terms of number of page faults per milliseconds

PFF algorithm

- Step 1 : the OS defines a system wide or per process critical page fault frequency, P.
- Step 2 : the OS measures and stores the time of the most recent page fault in the related PCB.
- Step 3 : when the page fault occurs, the OS acts as follows :
  - a) if the last page fault occurred less than T=1/p ms ago, the process is operating above the PFF threshold, and a new page frame is allocated to the process

PFF algorithm

  - b) otherwise, the process is operating below the PFF threshold P, and a page frame occupied by a page whose reference bit and written-into bit are not set is freed to accommodate the new page.
  - c) the OS sweeps and resets reference bits of all resident pages. Pages that are found to be unused, unmodified and not shared since the last sweep are released and the freed page frames are returned to the free pool for future use.
- PFF may be implemented as a Global or Local policy.

Working Set Theory

- Based on the assumption of locality; takes into consideration, the interactive nature of page replacement and allocation policies.
- Its three basic premises are
  - During any interval of time, a running process favors a subset of its pages
  - The memory reference patterns of a process exhibit a high correlation between the immediate past and the immediate future.
  - The frequency with which a given page is referenced is a slowly changing function of time.

Working Set Theory

- The working set of a program is defined to be the set of pages referenced by the program during a recent interval of time. As the program moves from one locality to another in the course of its execution, the no. and identities of the pages in the working set changes accordingly.
- A program's working set at the time of the t^{th} memory reference is

  $$\mathbf{W(t, \theta) = \{ i \in N \mid page\ i\ appears\ among\ r_{t-\theta+1} \ldots r_t \}}$$

  Where r_t is the memory reference at time t.
- Thus, **W(t, θ)** is the set of pages referenced during the last **θ** memory references. As the program executes, both the size of its working set and the identity of the pages that it comprises vary with time.

Working Set Theory

- The working set principle states that
  - A program should be run if and only if its working set is in memory
  - A page may not be removed from the memory if it is the member of the working set of a running program
- An important parameter for implementation of working set principle is the size of the time window . If it is too short, the working set may not fully encompass the current locality and hence cause an increased no. of page fault. On the other hand, if it is too long, it may span more than one locality and thus absorb more page frames than necessary.
- Working set theory prevents thrashing while maintaining a high degree of multiprogramming, thus optimizes CPU utilization.

IMPORTANT CONSIDERATIONS

Pre-paging

- Pre-paging is a technique that prevents the high level of initial paging i.e. the large no. of page faults that occur when a process is started or restarted after being swapped-in.
- The strategy is to bring into memory at one time all the pages that will be needed.
- An important question in pre-paging is whether the cost of using it is less than the cost of servicing the corresponding page faults. Assume that ***s*** pages are pre-paged and a fraction ***α*** of these ***s*** pages are actually used. The question is whether the cost of the ***s * α*** saved page faults is greater or less than the cost of pre-paging ***s * (1 - α)*** unnecessary pages. If ***α*** is close to zero, pre-paging loses; if it is close to one, pre-paging wins.

Page size

- Decreasing the page size, increases the size of PMT. Since each process must have a copy of its PMT, larger page size is desirable
- Memory is better utilized with smaller pages; reduces internal/page fragmentation.
- Consider the time required to read or write a page. I/O time is composed of seek, latency, and transfer times. With a page size of 512 bytes and a transfer rate of 2MB /sec, the transfer time is 0.2 ms, latency is 8 ms and seek time is 20 ms. Thus total I/O time is 28.2 ms. To transfer two such pages, the time requited would be 56.4 ms. However, if we double the page size I/O time would be 28.4 ms. Thus, a desire to minimize I/O urges for a larger page size.

Page size

- A smaller page size allows each page to match program locality more accurately, thus we have a better resolution, allowing us to isolate only the memory that is actually needed. With larger page size, we must allocate and transfer not only what is needed, but also anything else that happens to be in page, whether it is needed or not. Thus, a smaller page size should result in less I/O and less total allocated memory
- On the other hand, too smaller page sizes increases the page faults. Each page fault generates the large amount of overhead needed for processing the interrupts, saving registers, replacing a page, queuing the page device, and updating tables. To minimize the page faults, we need to have a large page size.

Page size

- There is no better solution for the page size problem. Some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size.
- The historical trend is towards larger page size.
- In modern computers, page size varies from 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes.

I/O Interlock

- When using demand paging, some of the pages needs to be locked in memory, particularly in situations when I/O is done from user memory. If the page containing buffer for the waiting process gets replaced, the I/O occurs at the specified address which is now occupied by a different page.
- Solution 1 : I/O to user memory should be done through System memory; results in high overhead.
- Solution 2 : critical pages which should not get replaced should be locked