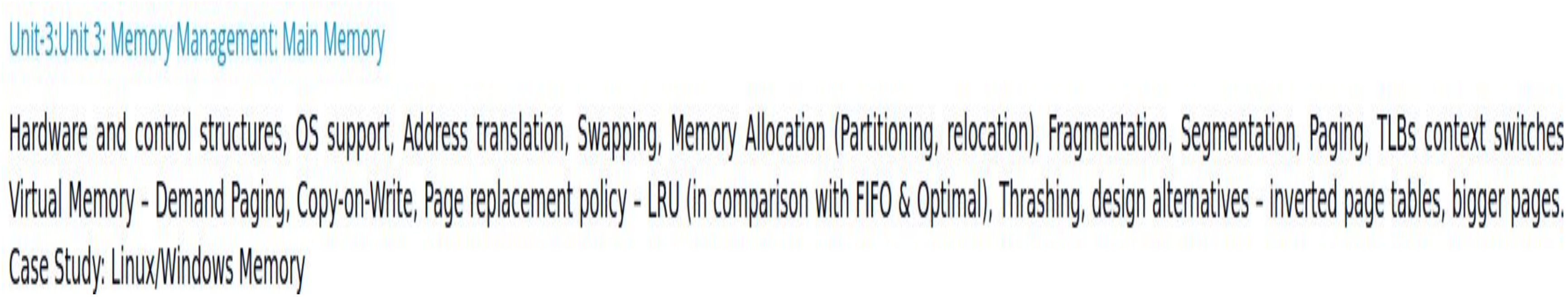
OPERATING SYSTEMS

Memory Management - 5

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

Course Syllabus - Unit 3



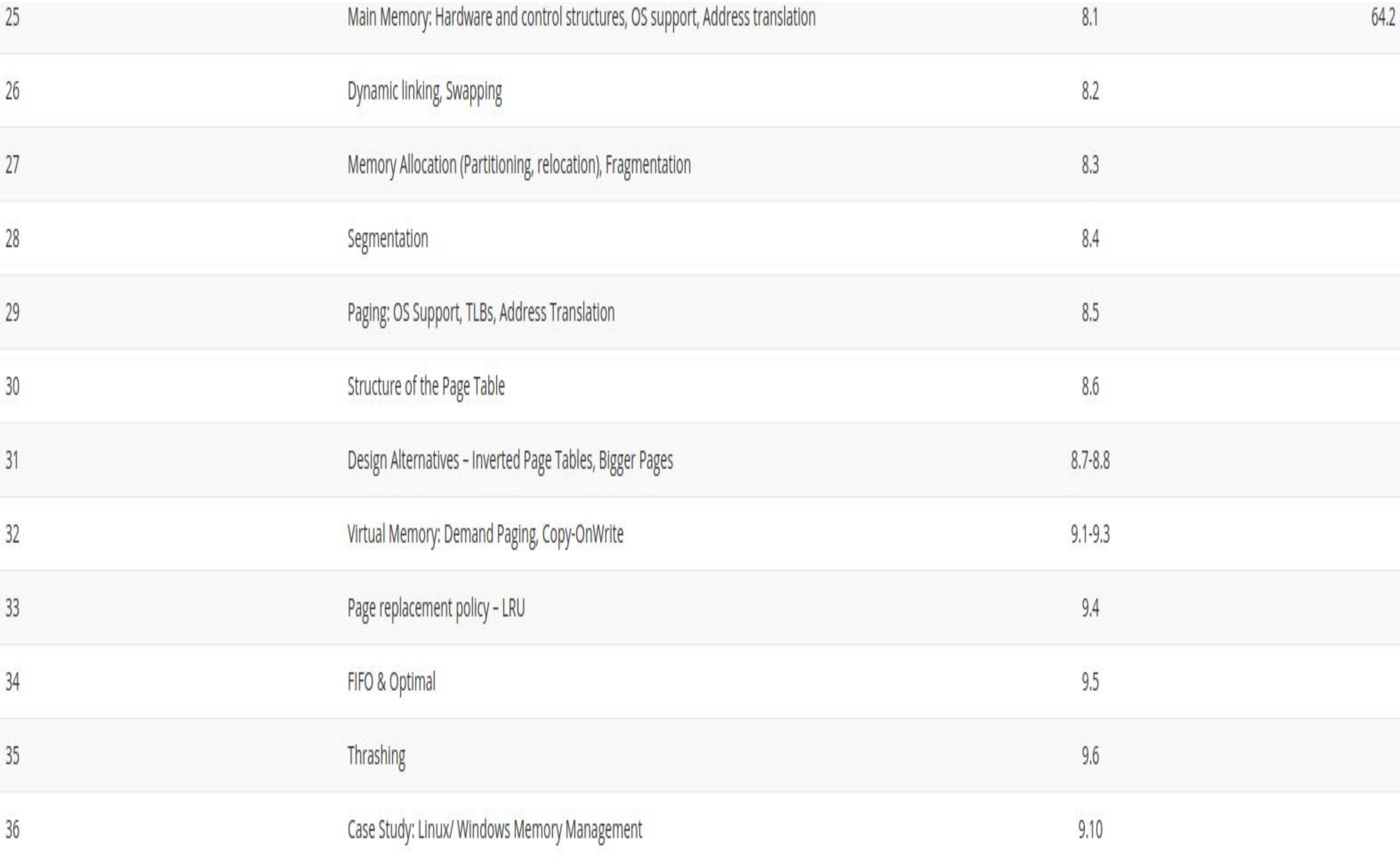
Unit-3:Unit 3: Memory Management: Main Memory

Hardware and control structures, OS support, Address translation, Swapping, Memory Allocation (Partitioning, relocation), Fragmentation, Segmentation, Paging, TLBs context switches

Virtual Memory - Demand Paging, Copy-on-Write, Page replacement policy - LRU (in comparison with FIFO & Optimal), Thrashing, design alternatives - inverted page tables, bigger pages.

Case Study: Linux/Windows Memory

Course Outline



25	Main Memory: Hardware and control structures, OS support, Address translation	8.1	64.2
26	Dynamic linking, Swapping	8.2	
27	Memory Allocation (Partitioning, relocation), Fragmentation	8.3	
28	Segmentation	8.4	
29	Paging: OS Support, TLBs, Address Translation	8.5	
30	Structure of the Page Table	8.6	
31	Design Alternatives - Inverted Page Tables, Bigger Pages	8.7-8.8	
32	Virtual Memory: Demand Paging, Copy-OnWrite	9.1-9.3	
33	Page replacement policy - LRU	9.4	
34	FIFO & Optimal	9.5	
35	Thrashing	9.6	
36	Case Study: Linux/ Windows Memory Management	9.10	

Topic Outline

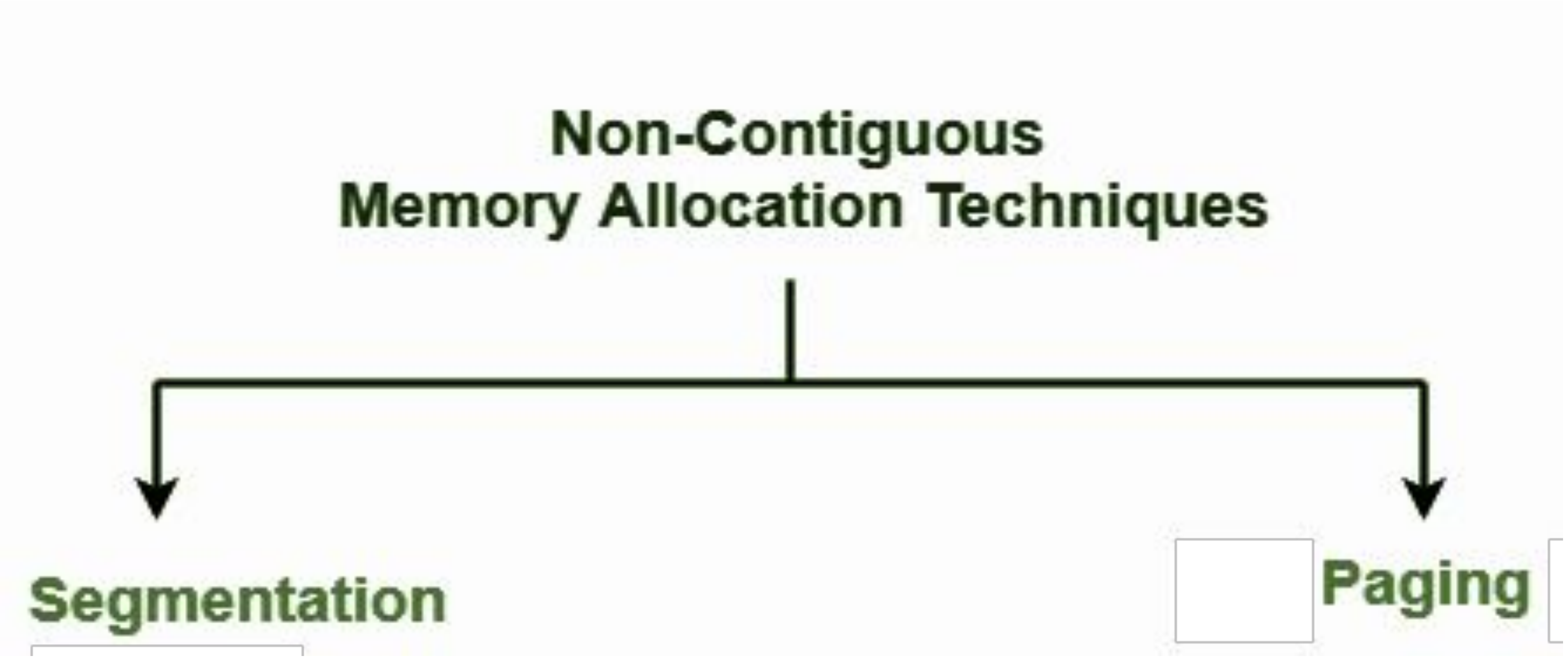
- Non Contiguous memory allocation
- Paging
- Address Translation Scheme
- Paging Hardware
- Paging model of Logical and Physical Memory
- Paging Example
- Free Frames

Topic Outline

- Implementation of Page Table
- Associative Memory
- Table Lookaside Buffer
- Effective Access Time
- Memory Protection
- Valid - Invalid Bit
- Shared Pages
- Shared Pages Example

Non - Contiguous Memory Allocation

- Non-contiguous memory allocation is a memory allocation technique.
- It allows to store parts of a single process in a non-contiguous fashion.
- Thus, different parts of the same process can be stored at different places in the main memory.



Non - Contiguous Memory Allocation

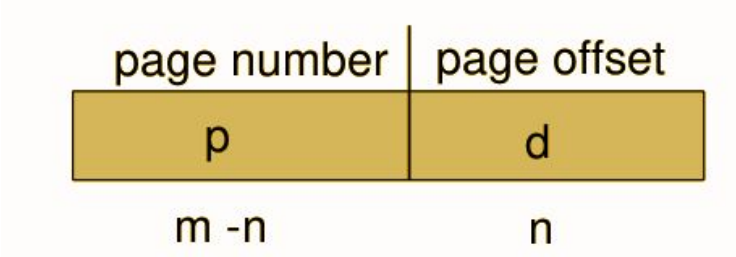
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks

Paging

- Divide physical memory into fixed-sized blocks called **Frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **Pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a page table to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Address Translation Scheme

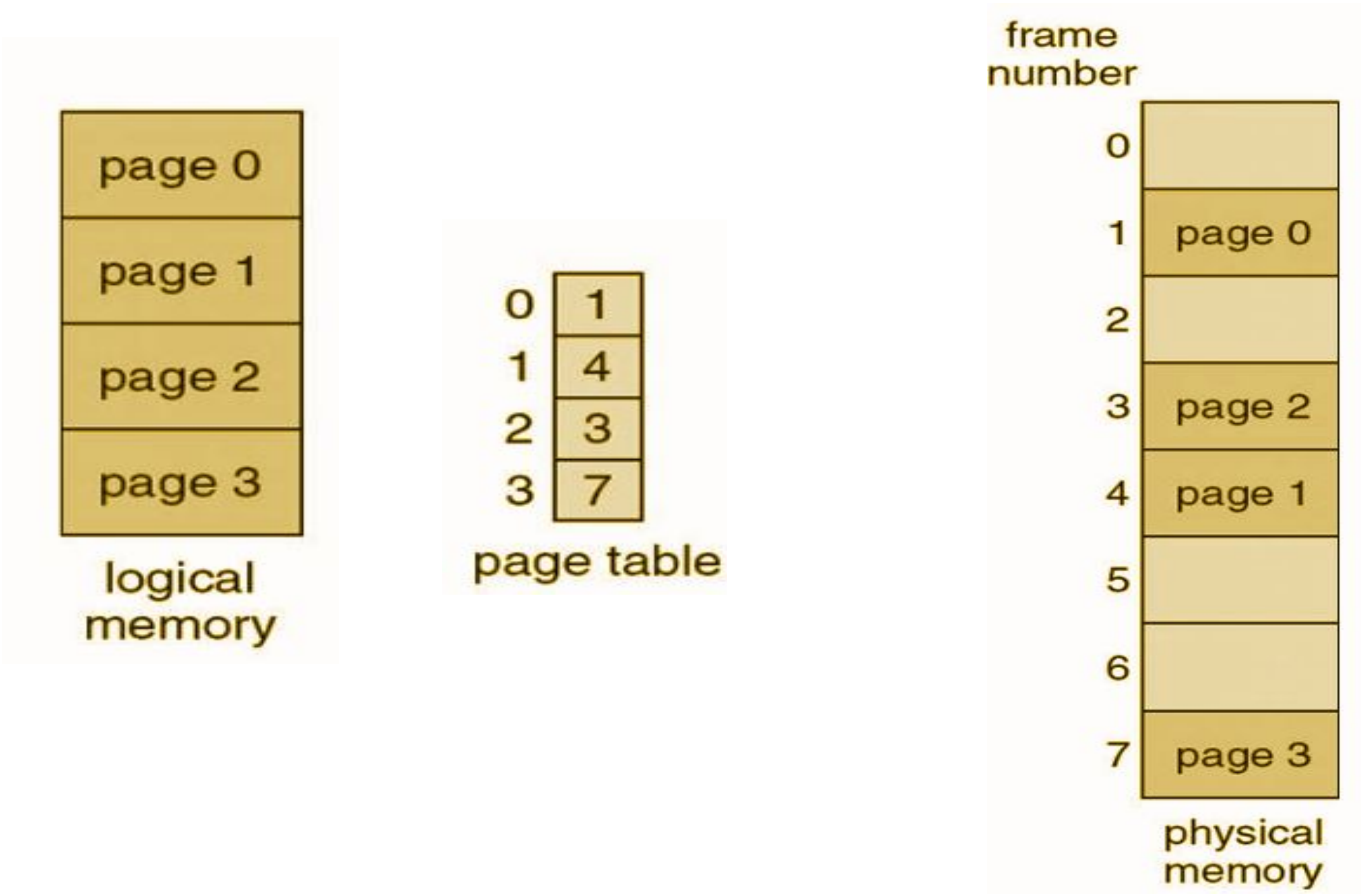
- Address generated by CPU is divided into:
  - **Page number (p)** => used as an index into a page table which contains base address of each page in physical memory, which is nothing but the frame address
  - **Page offset (d)** => combined with base address to define the physical memory address that is sent to the memory unit



page number	page offset
p	d
m -n	n

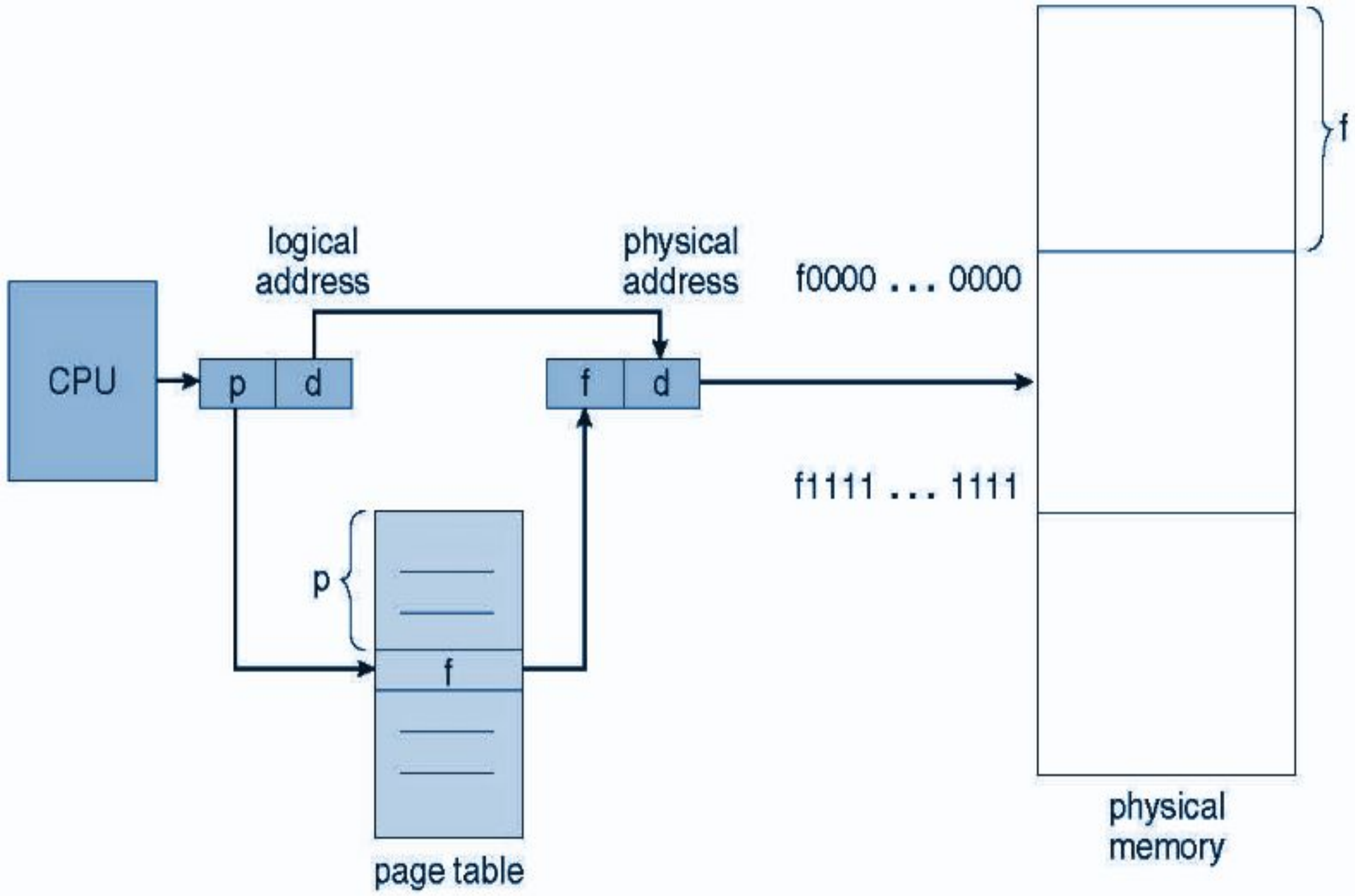
- For given logical address space 2**m and page size 2**n

Paging Hardware



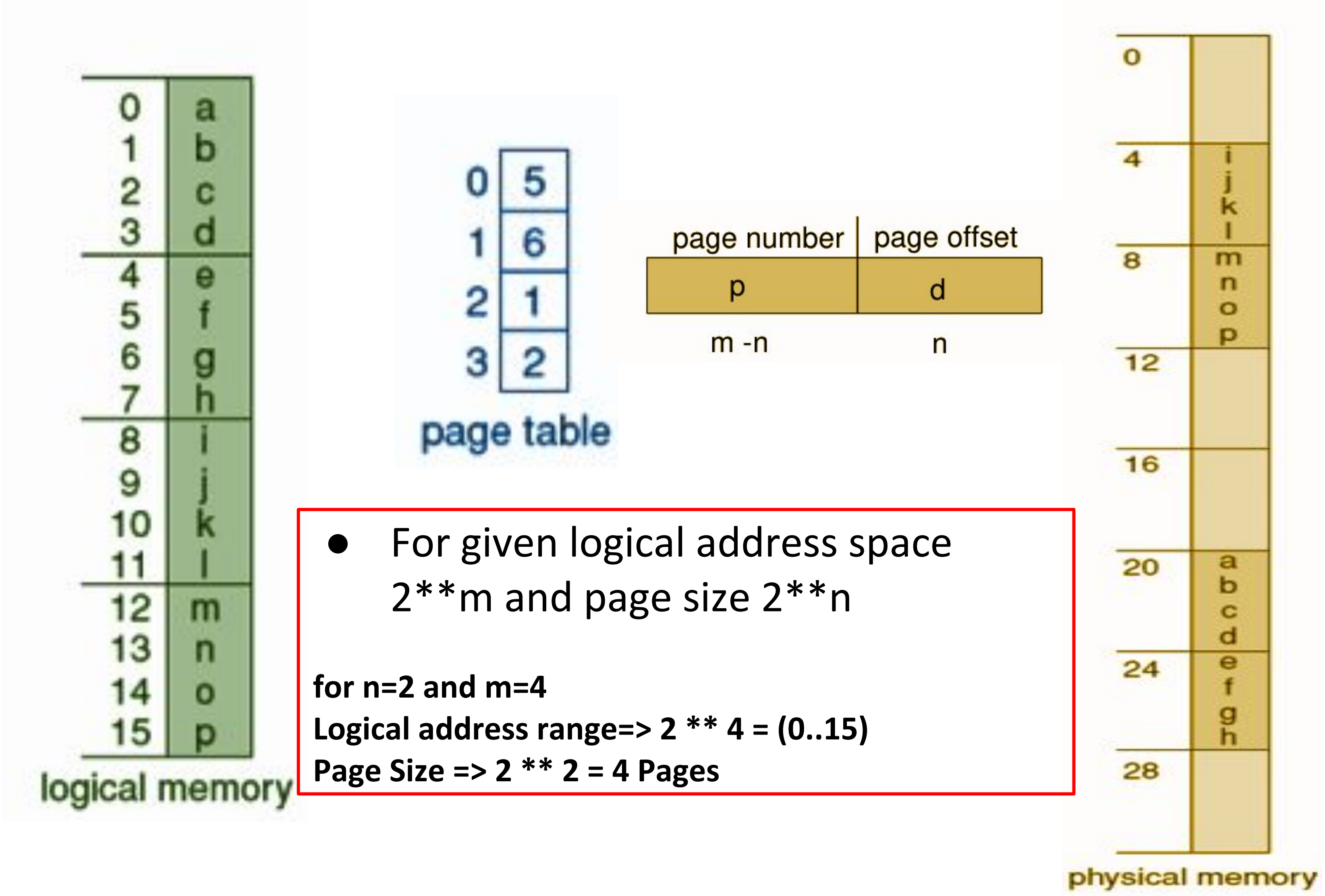
Slides Adapted from Operating System Concepts 9/e © Authors

Paging Model of Logical and Physical Memory



Slides Adapted from Operating System Concepts 9/e © Authors

Paging Example

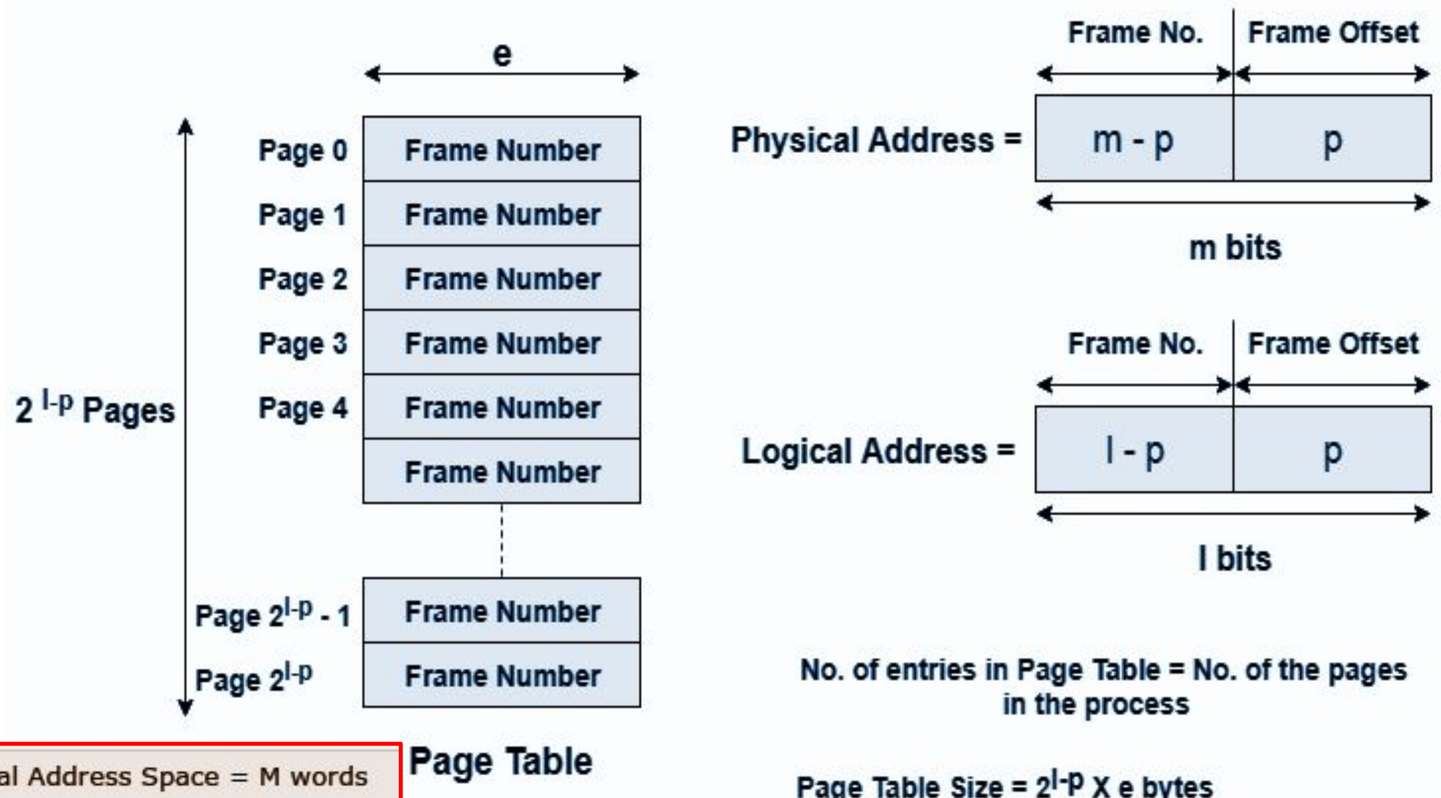


- For given logical address space 2**m and page size 2**n

for n=2 and m=4
Logical address range=> 2 ** 4 = (0..15)
Page Size => 2 ** 2 = 4 Pages

Slides Adapted from Operating System Concepts 9/e © Authors

Paging Example



No. of entries in Page Table = No. of the pages in the process

Page Table Size = 2^{l-p} X e bytes

e = m-p (Frame Size) bits

Physical Address Space = M words

Logical Address Space = L words

Page Size = P words

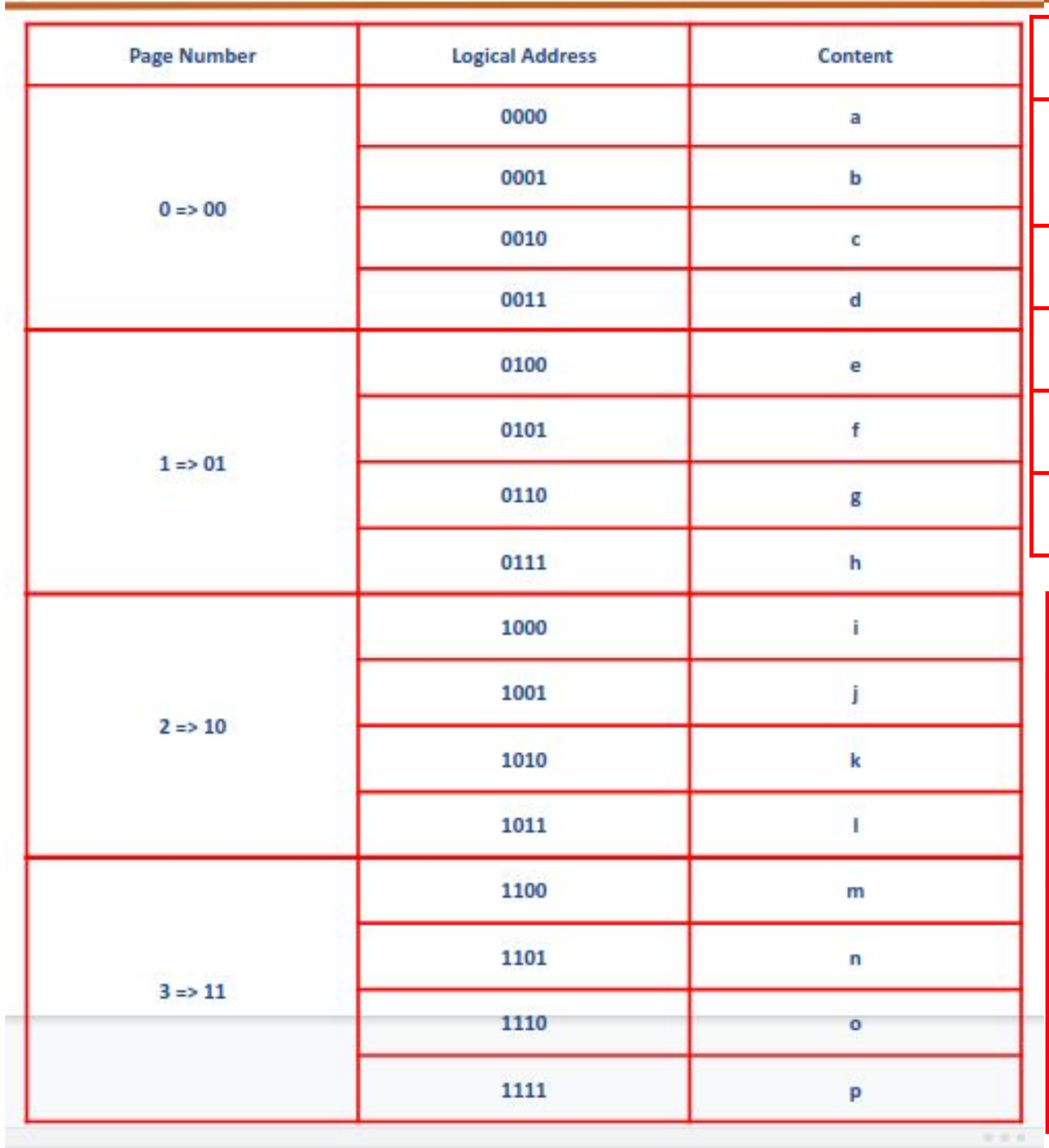
Physical Address = $\log_2 M$ = m bits

Logical Address = $\log_2 L$ = l bits

page offset = $\log_2 P$ = p bits

https://www.kernel.org/doc/gorman/html/understand/understand006.html

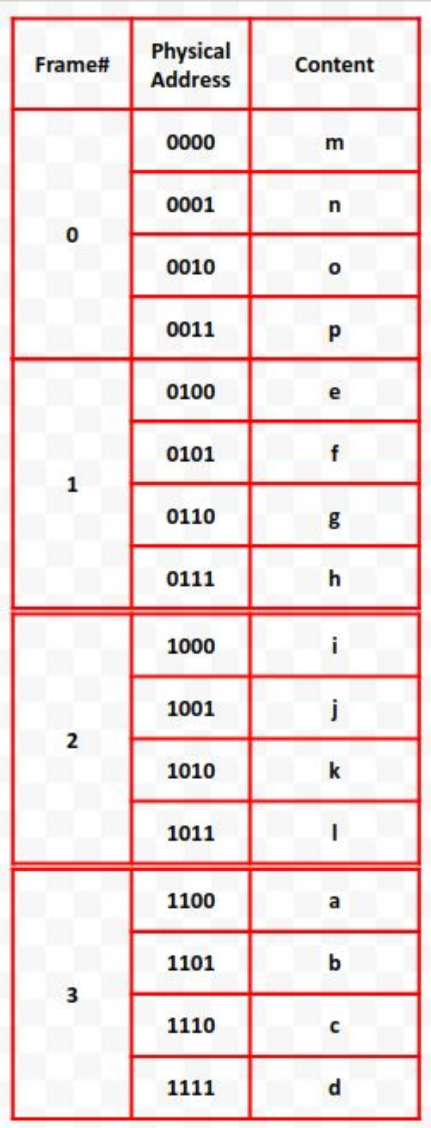
Paging Example



Page Number	Logical Address	Content
0 => 00	0000	a
	0001	b
	0010	c
	0011	d
1 => 01	0100	e
	0101	f
	0110	g
	0111	h
2 => 10	1000	i
	1001	j
	1010	k
	1011	l
3 => 11	1100	m
	1101	n
	1110	o
	1111	p

Page Table	
Page Number	Frame Number
0	3
1	1
2	2
3	0

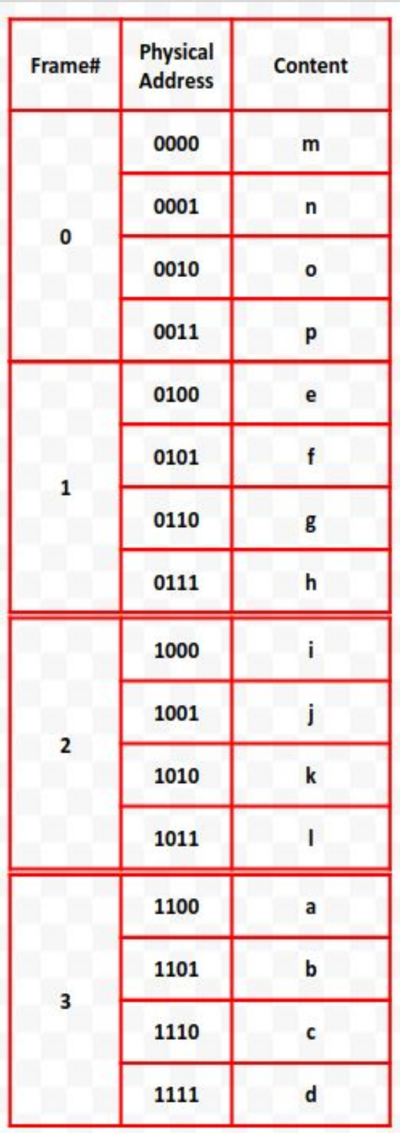
For Logical address	
Logical Address	Page #
1001	2
1100	3
0001	0
0101	1



Frame#	Physical Address	Content
0	0000	m
	0001	n
	0010	o
	0011	p
1	0100	e
	0101	f
	0110	g
	0111	h
2	1000	i
	1001	j
	1010	k
	1011	l
3	1100	a
	1101	b
	1110	c
	1111	d

Paging Example

Logical Address generated by CPU	Physical Address generated in runtime
1010 =>[1010] =>(10,10) => (Page 2, Offset 2) => k	(Frame 2, Offset 2) => (1000,10) => 1010 => [1010] => k
0011 => [0011] => (00,11) => (Page 0, Offset 3) => d	(Frame 3, Offset 3) => (1100,11) => 1111 => [1111] => d
1101 => [1101]=>(11,01) =>(Page3, offset 1) =>n	(Frame 0, Offset 1) => (0000, offset 1) =>0001 => [0001] => n
1111 => ? => ? => ? => ?	(Frame ?, Offset ?) => ? => ? => ? => ?



Frame#	Physical Address	Content
0	0000	m
	0001	n
	0010	o
	0011	p
1	0100	e
	0101	f
	0110	g
	0111	h
2	1000	i
	1001	j
	1010	k
	1011	l
3	1100	a
	1101	b
	1110	c
	1111	d

For Logical address	
Logical Address	Page #
1001	2
1100	3
0001	0
0101	1

Page Table	
Page Number	Frame Number
0	3
1	1
2	2
3	0

Paging : Calculating Internal Fragmentation

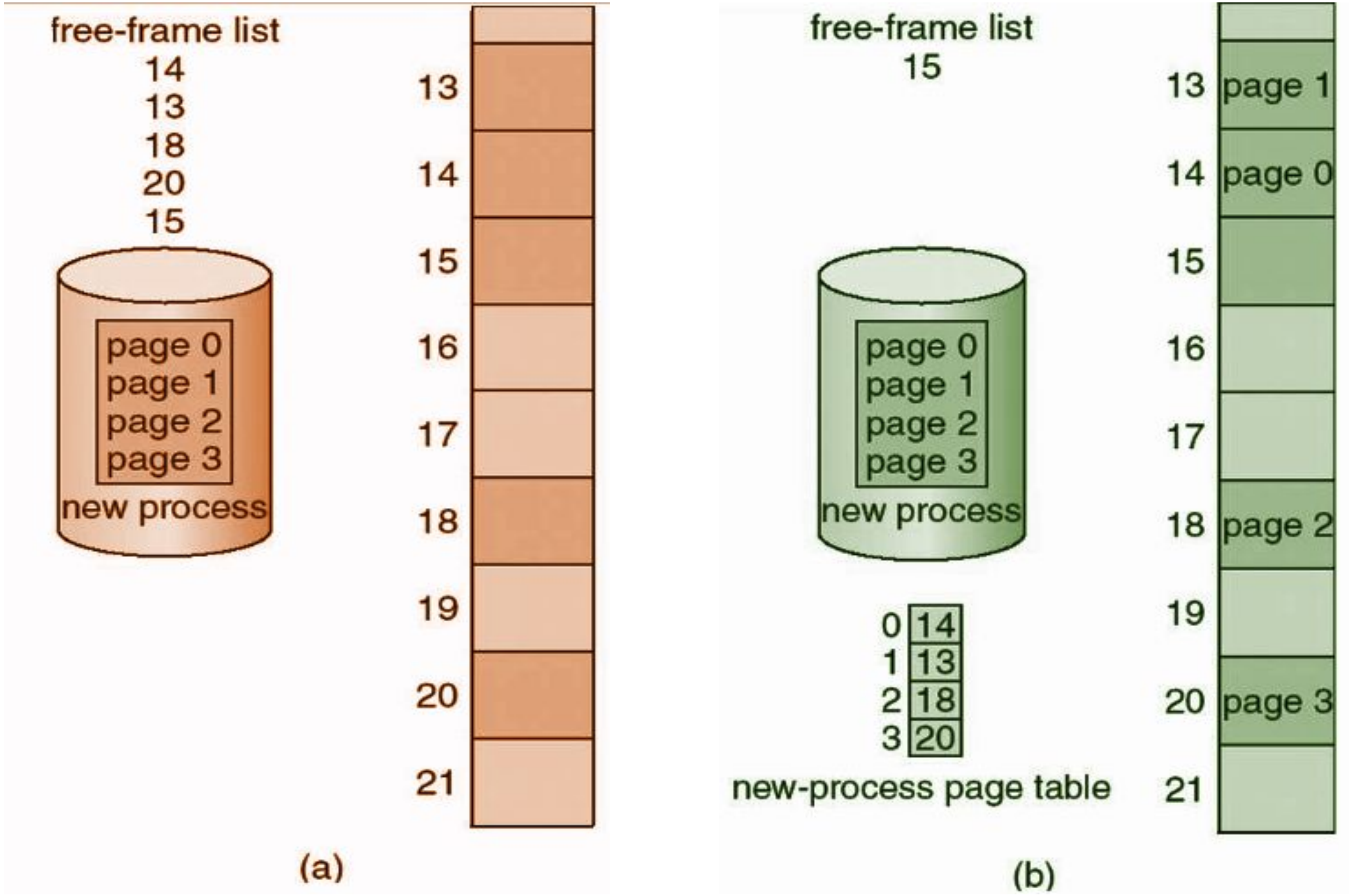
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable ?
- But each page table entry takes memory to track
- Page sizes growing over time

Paging : Calculating Internal Fragmentation

- Process view and physical memory now very different

- By implementation process can only access its own memory

Free Frame List



Slides Adapted from Operating System Concepts 9/e © Authors

Implementation of Page table

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) 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 Lookaside Buffers (TLBs)**

Implementation of Page table

- Some TLBs store **Address-Space Identifiers (ASIDs)** in each TLB entry that uniquely identifies each process to provide address-space protection for that process

- Otherwise need to flush at every context switch

- TLBs typically small (64 to 1,024 entries)

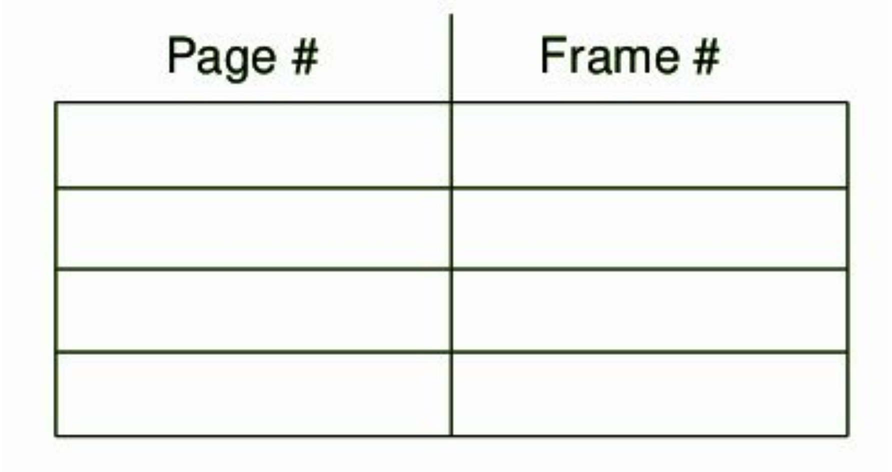
- On a TLB miss, value is loaded into the TLB for faster access next time

- Replacement policies must be considered

- Some entries can be wired down for permanent fast access

Associative Memory or Translation Lookaside Buffers (TLBs)

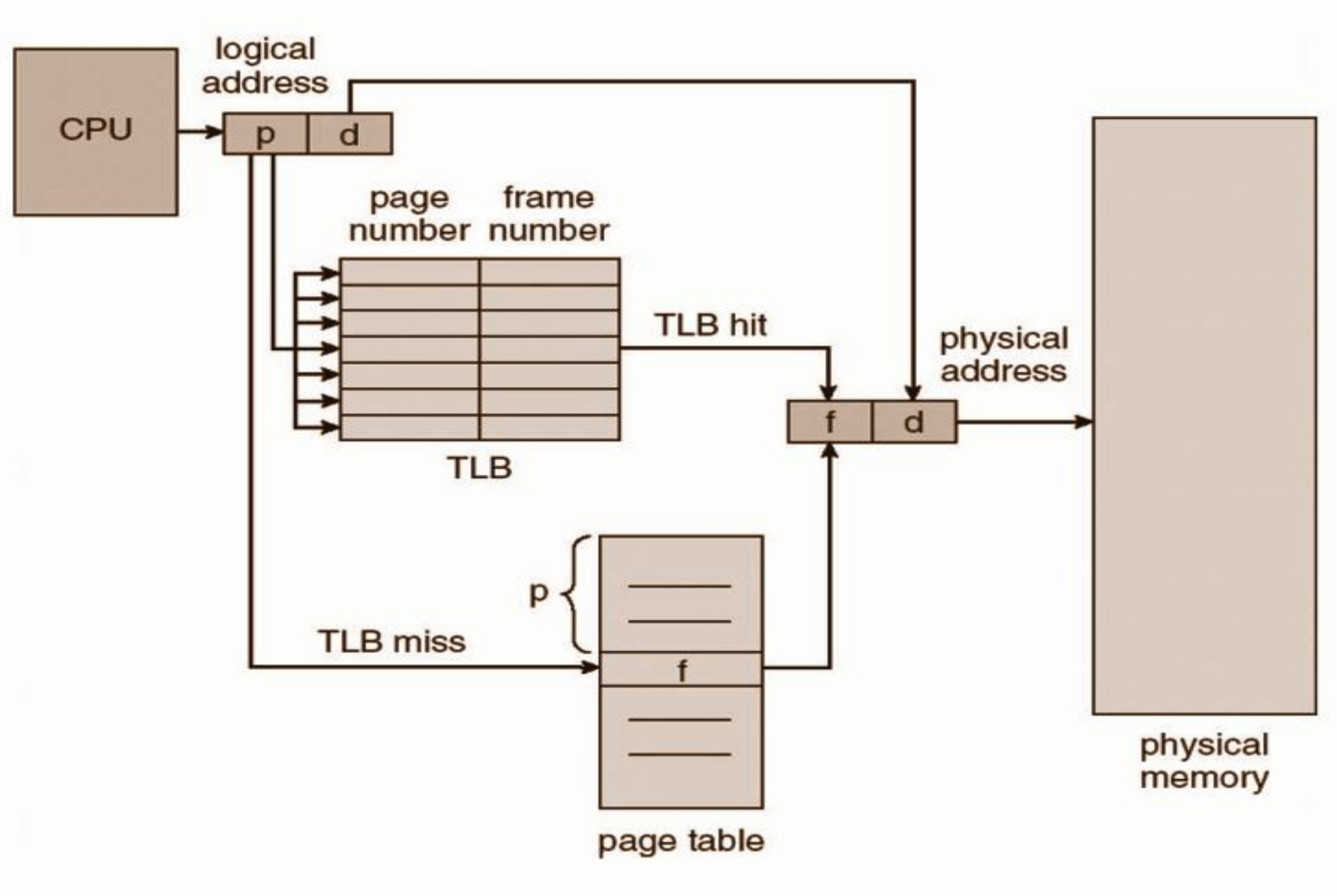
- Associative memory – parallel search



Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware with TLB



Slides Adapted from Operating System Concepts 9/e © Authors

Effective Access Time

- Associative Lookup = time unit
  - Can be < 10% of memory access time

- Hit ratio = α
- TLB search and access time => ε (Epsilon)

- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Slides Adapted from Operating System Concepts 9/e © Authors

Effective Access Time

- Effective Access Time (EAT)
  - **EAT = $(1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$**
    $= 2 + \varepsilon - \alpha$
  - **EAT = Found + NotFound (Considering Epsilon)**
  - **Found=>** α X 120 ns => 0.8 X 120 => 96 ns
  - **Not Found=>** (1- α) X 200ns => 0.2 X 200 ns=>40 ns

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - **EAT = 0.80 x 120 + 0.20 x 200 = 136 ns (Considering Epsilon)**

- Consider more realistic hit ratio => 99%, ε = 20 ns for TLB search, 100ns for memory access
  - **EAT = 0.99 x 120 + 0.01 x 200 = 119 ns (Considering Epsilon)**

Effective Access Time

- Effective Access Time (EAT)
  - **EAT = $(1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$**
    $= 2 + \varepsilon - \alpha$
  - **EAT = Found + NotFound (ignoring Epsilon)**
  - **Found=>** α X 100ns => 0.8 X 100 => 80ns
  - **Not Found=>** (1- α) X 200ns => 0.2 X 200 ns=>40ns

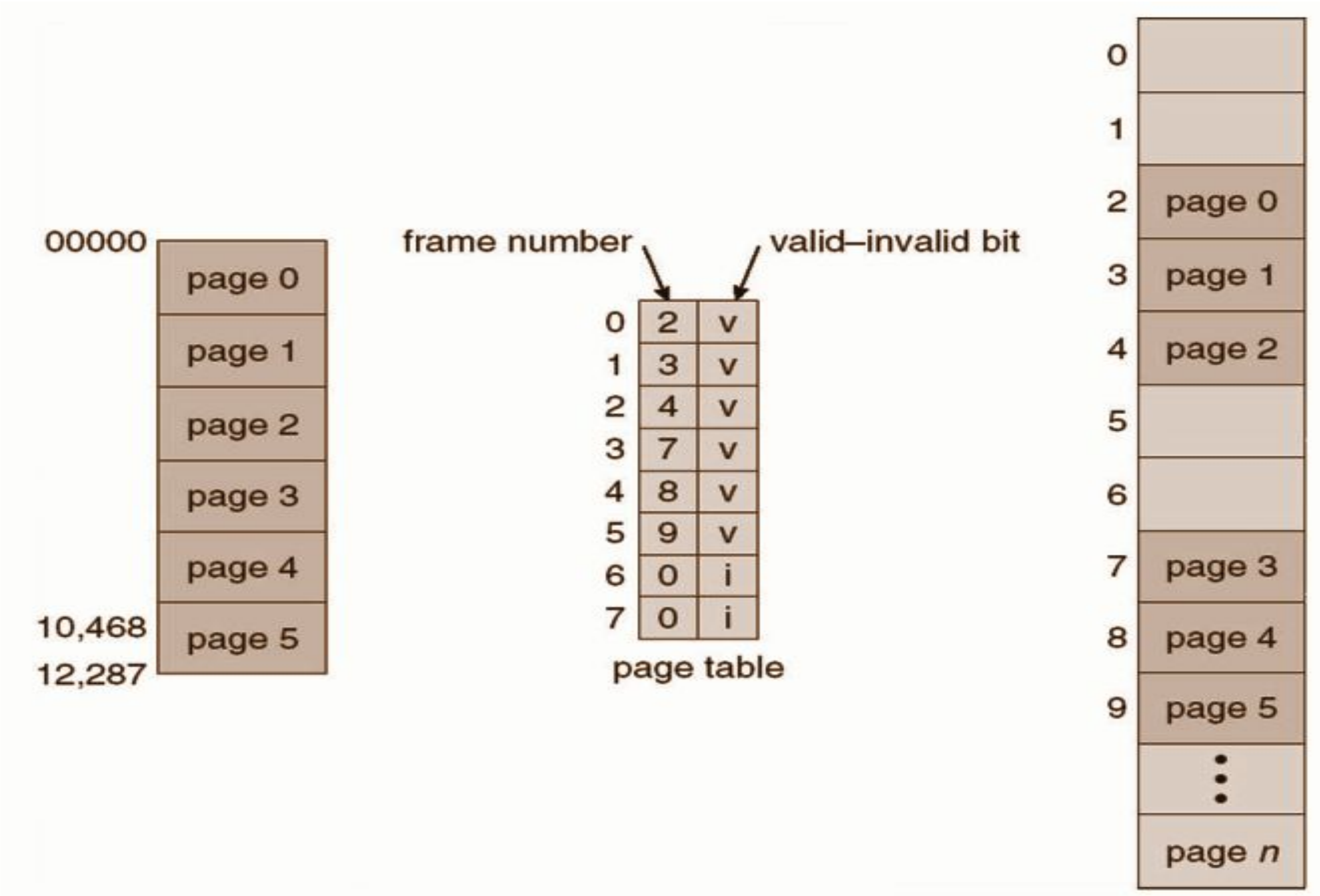
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - **EAT = 0.80 x 100 + 0.20 x 200 = 120ns (ignoring Epsilon)**

- Consider more realistic hit ratio => 99%, ε = 20ns for TLB search, 100ns for memory access
  - **EAT = 0.99 x 100 + 0.01 x 200 = 101ns (ignoring Epsilon)**

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- 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
  - Or use page-table length register (PTLR)
- Any violations result in a trap to the kernel

Valid Invalid Bit in the 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

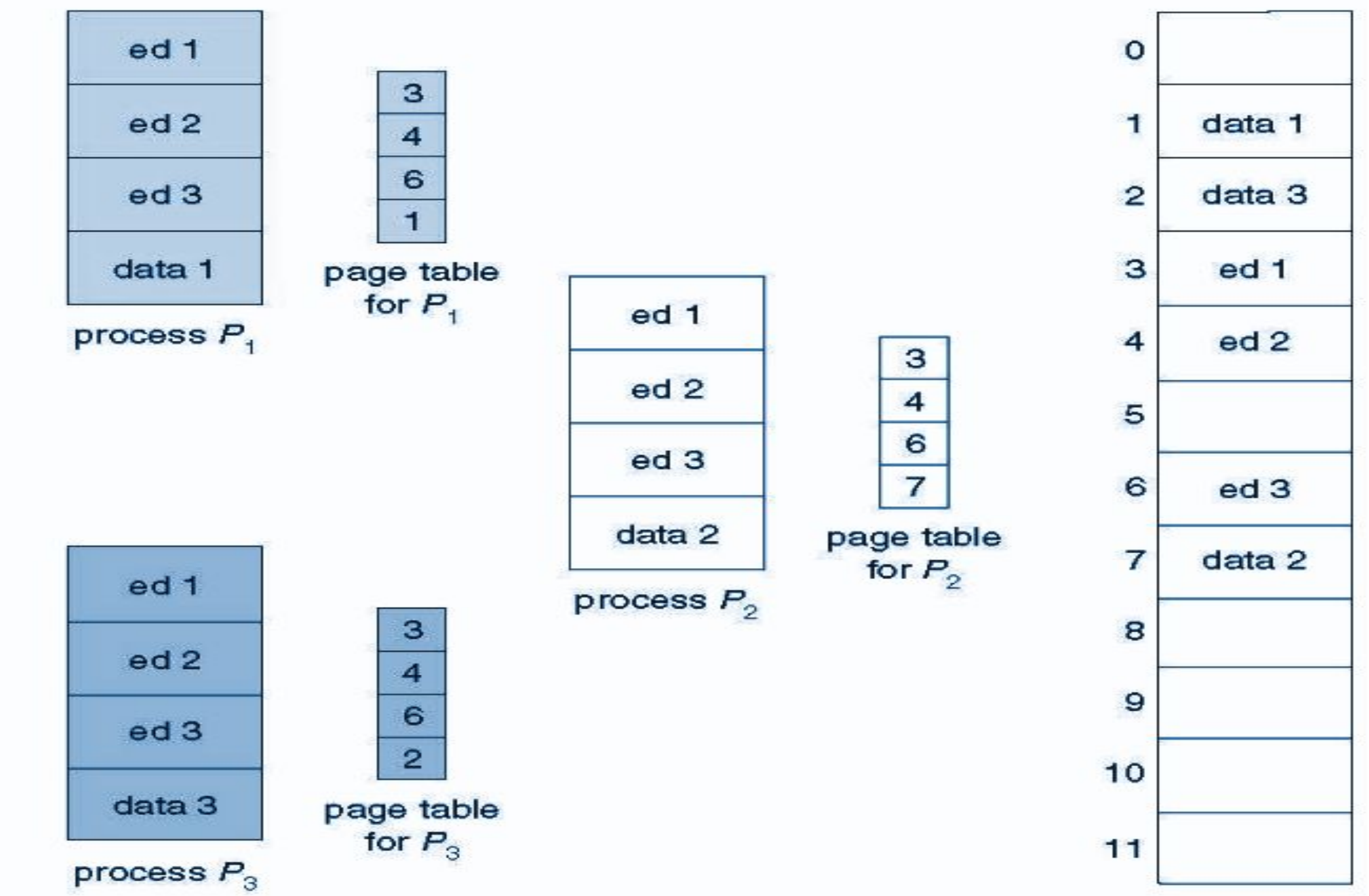
Slides Adapted from Operating System Concepts 9/e © Authors

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed

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

Shared Pages - Example



Slides Adapted from Operating System Concepts 9/e © Authors

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com and complete reading assignments provided by the Anchor on Edmodo