CS & IT ENGINEERING

Operating System

Memory Management (Part - 02)

Revision

Lecture No.- 10

By- Dr. Khaleel Khan Sir

Recap of Previous Lecture

Topic Basic Concepts of Memory Management

Topics to be Covered

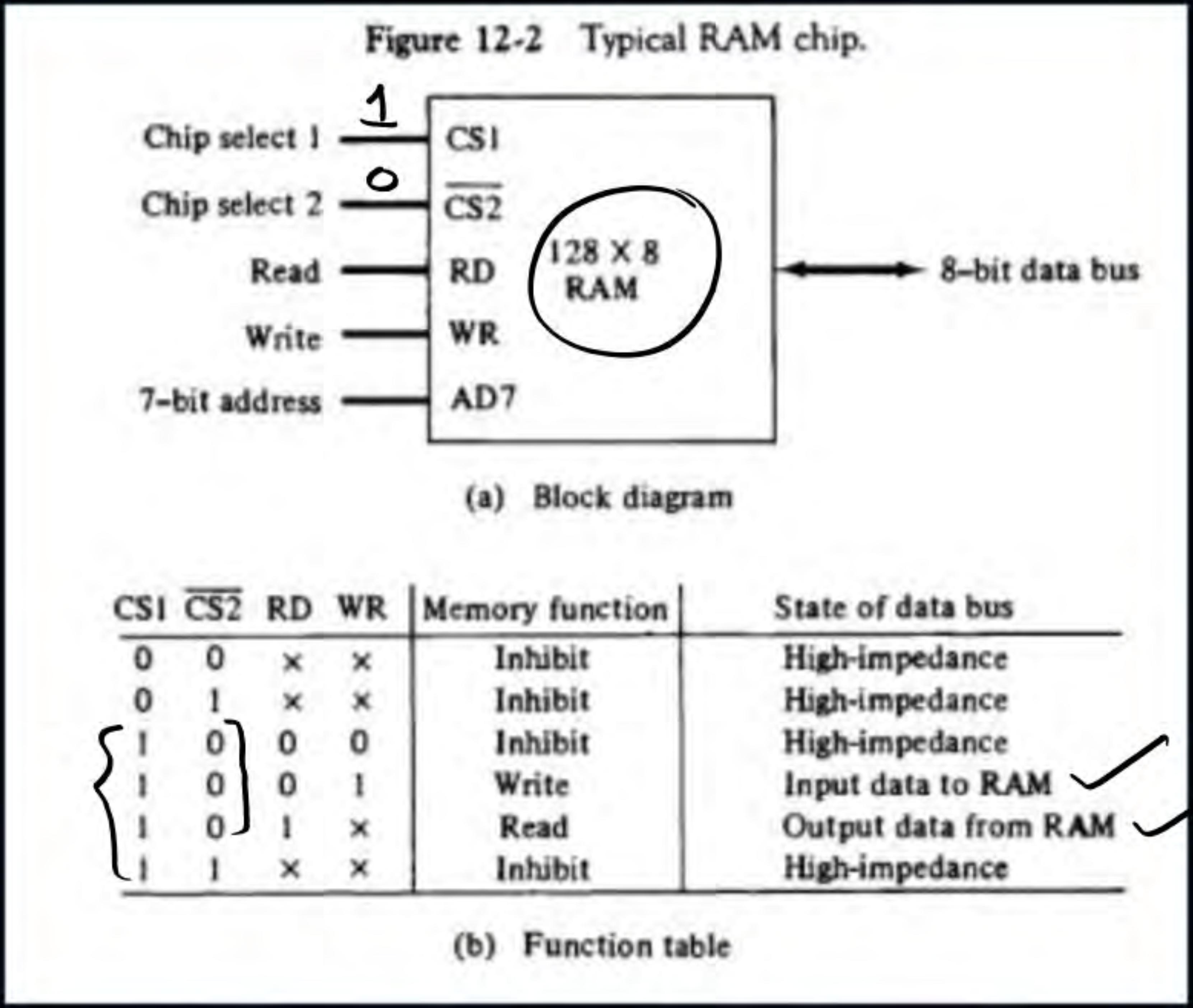
Topic Variable Partitions

Topic Protection

Topic Hardware Address Protection

Topic Paging Hardware, Multi Level Paging

Figure 12-2 Typical RAM chip.



(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM ✓
1	0	1	×	Read	Output data from RAM ✓
1	1	×	×	Inhibit	High-impedance

(b) Function table

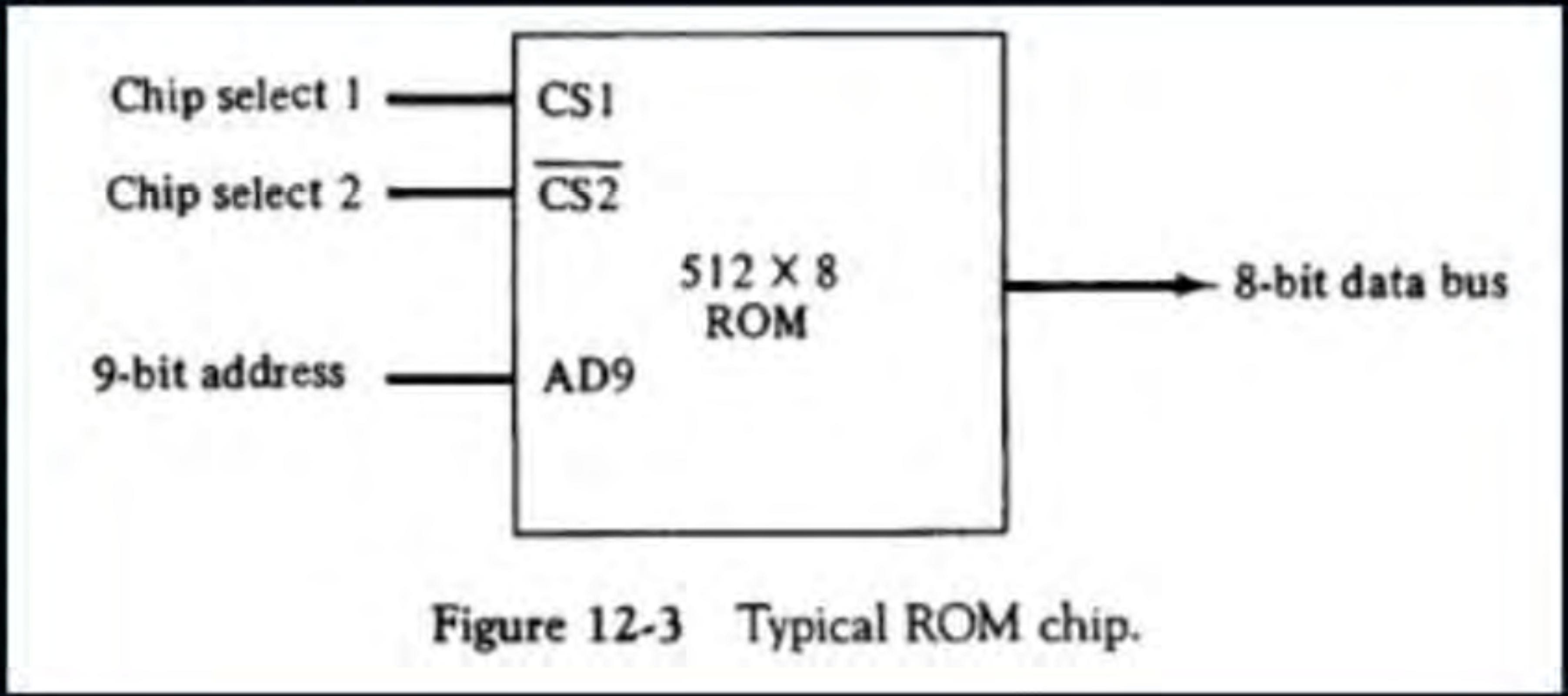


Figure 12-3 Typical ROM chip.

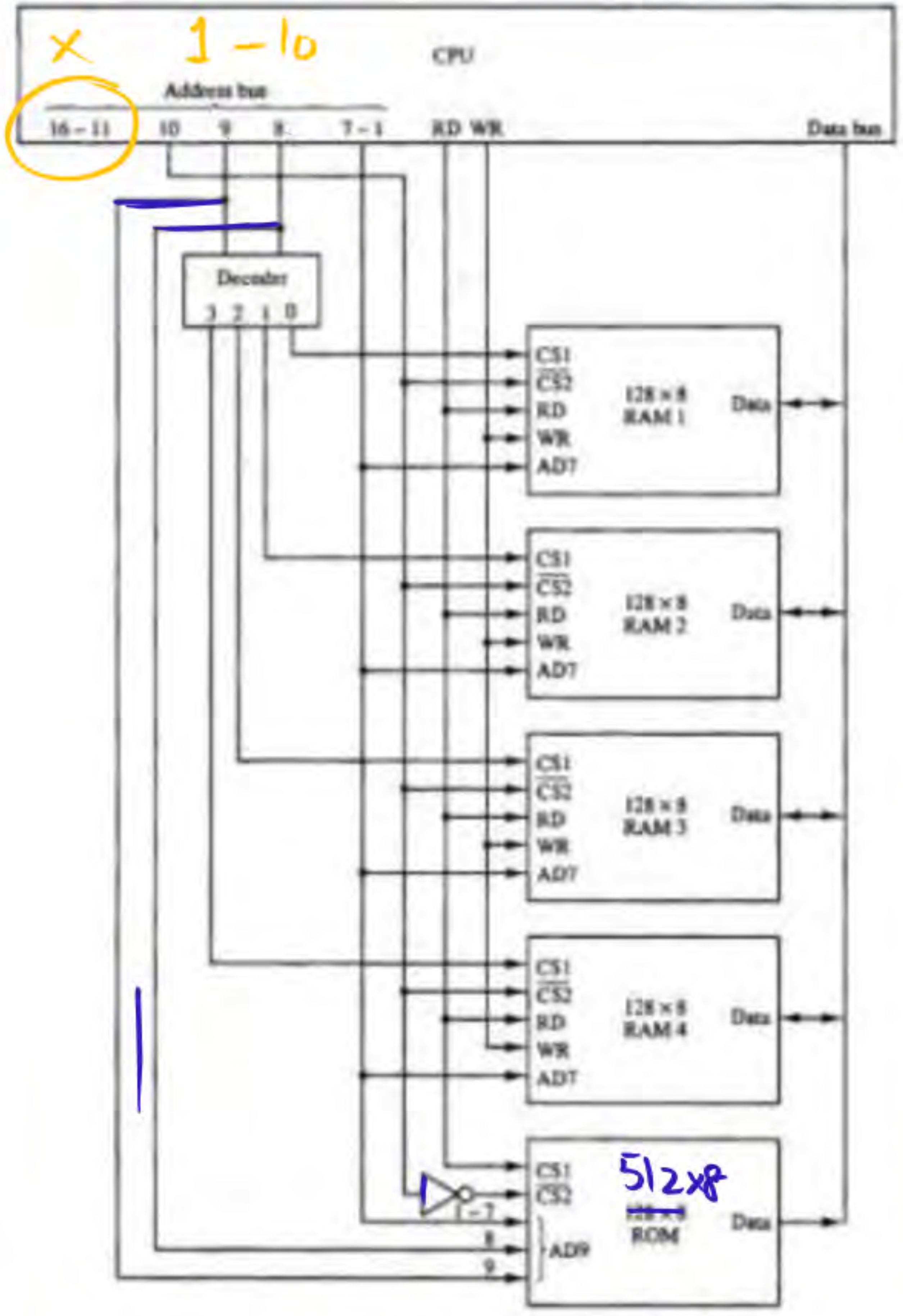


Figure 12-4 Memory connection to the CPU.

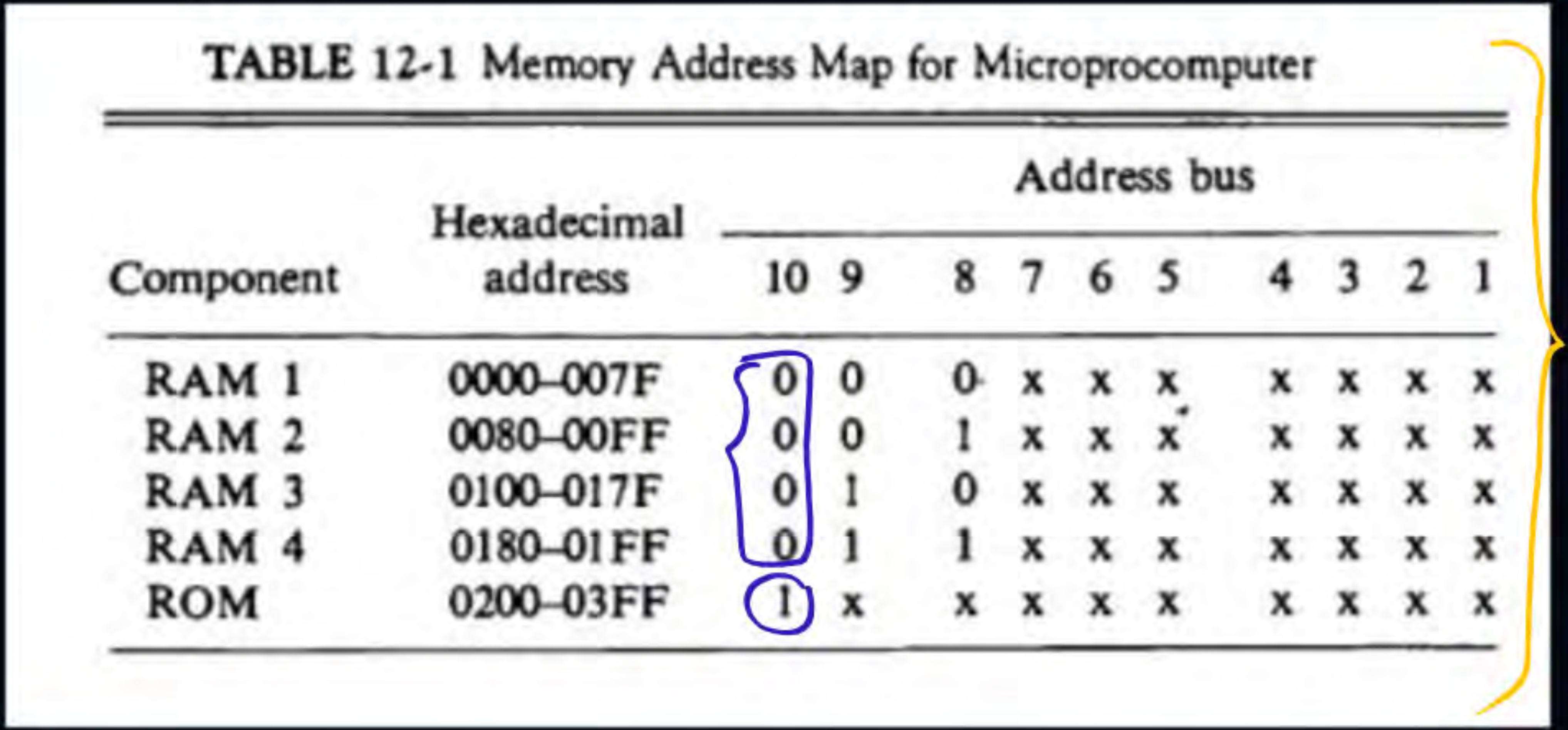


TABLE 12-1 Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

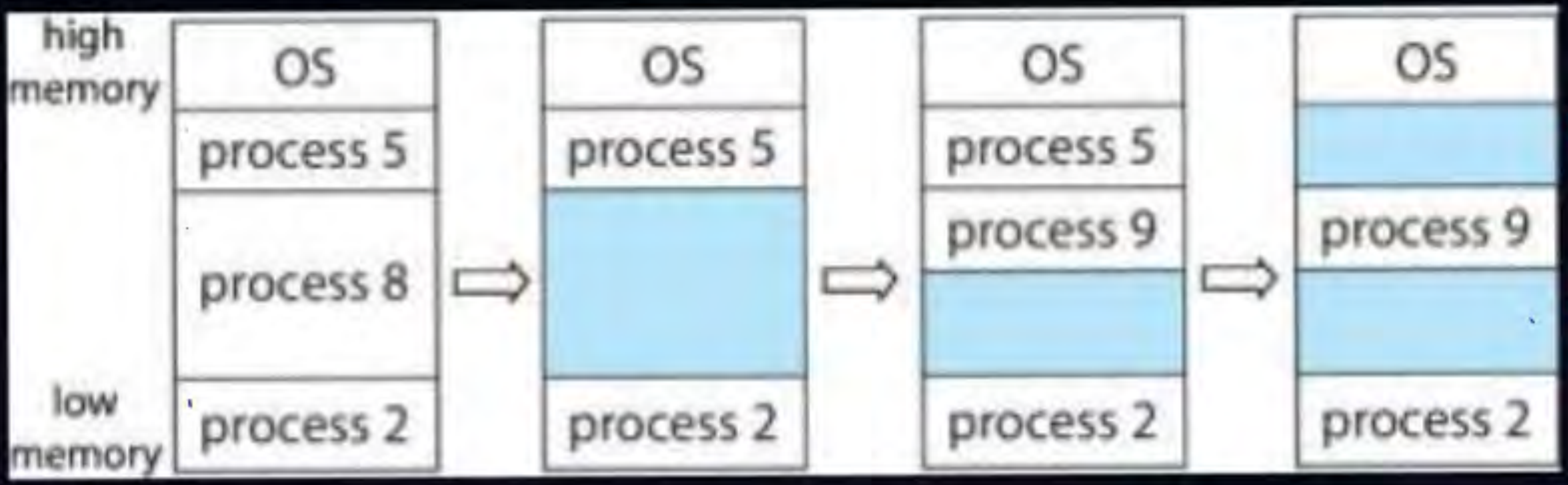
Topic : Variable Partition

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)

Coalescing

130K

free



50K

80K

Req: 85K

Topic : Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
- **First-fit:** Allocate the **first** 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

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

Topic : Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Topic : Fragmentation (Cont.)

- Reduce external fragmentation by **compaction** → Time consuming opn's
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

    R.T Address Binding
  - I/O problem
    - ✓ Latch job in memory while it is involved in I/O
    - ✓ Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?

Mem_Size = 2^{24} By

Part. Size = 2^{16} By

No. of Partitions = $\frac{2^{24}}{2^{16}} = 2^{8} = \underline{\underline{256}}$

Partition Address (pointer) = 8 bits = $\underline{1B}$

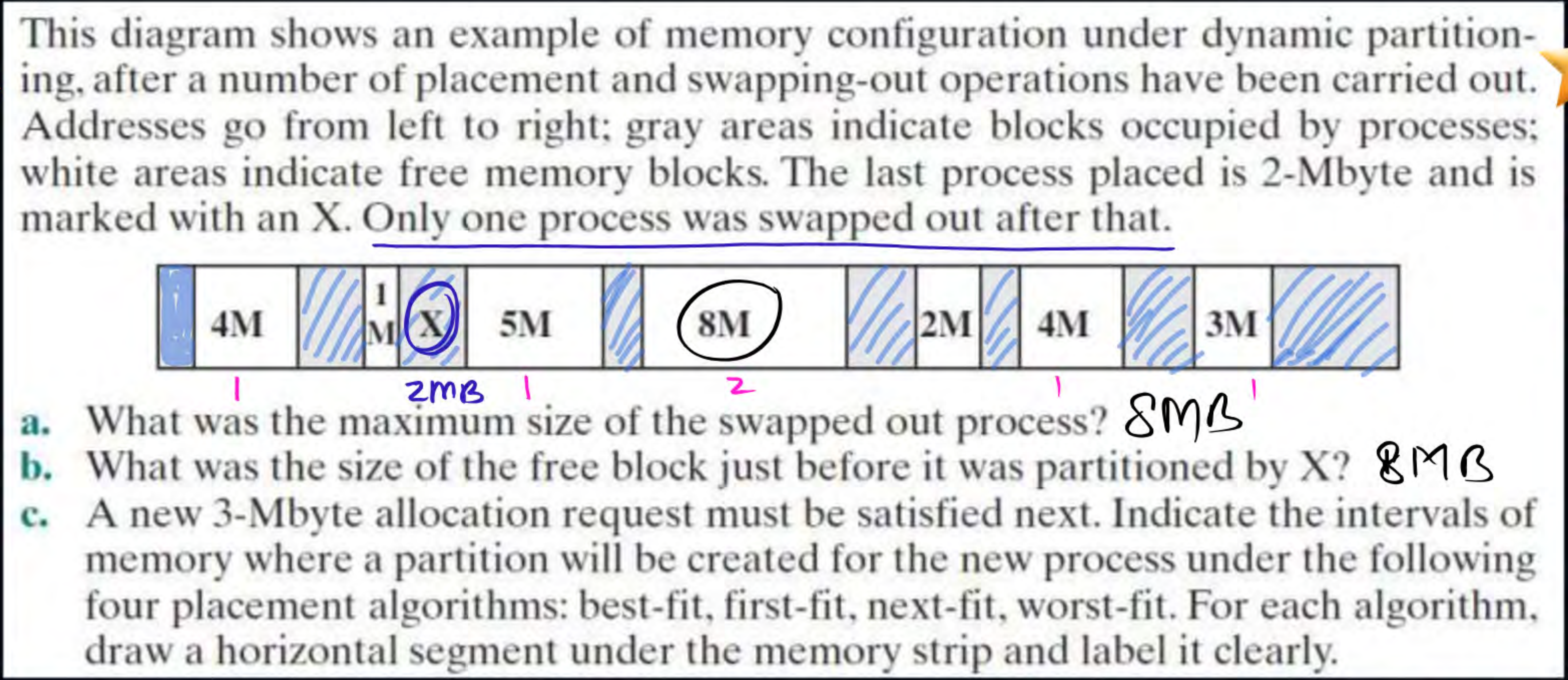
512 × 4B = $\underline{\underline{2KB}}$ ✓

512

Pid 3B	ptr 1B
⋮	⋮

$\underline{\underline{\text{P.T}}}$

→ If the System has 512 processes & Each process id in 3B; what is the Size of Process Table, if each entry contains Pid & ptr.



This diagram shows an example of memory configuration under dynamic partitioning, after a number of placement and swapping-out operations have been carried out. Addresses go from left to right; gray areas indicate blocks occupied by processes; white areas indicate free memory blocks. The last process placed is 2-Mbyte and is marked with an X. Only one process was swapped out after that.

▨	4M	▨	1M	X	5M	▨	8M	▨	2M	▨	4M	▨	3M	▨
	1			2MB	1		2				1		1	

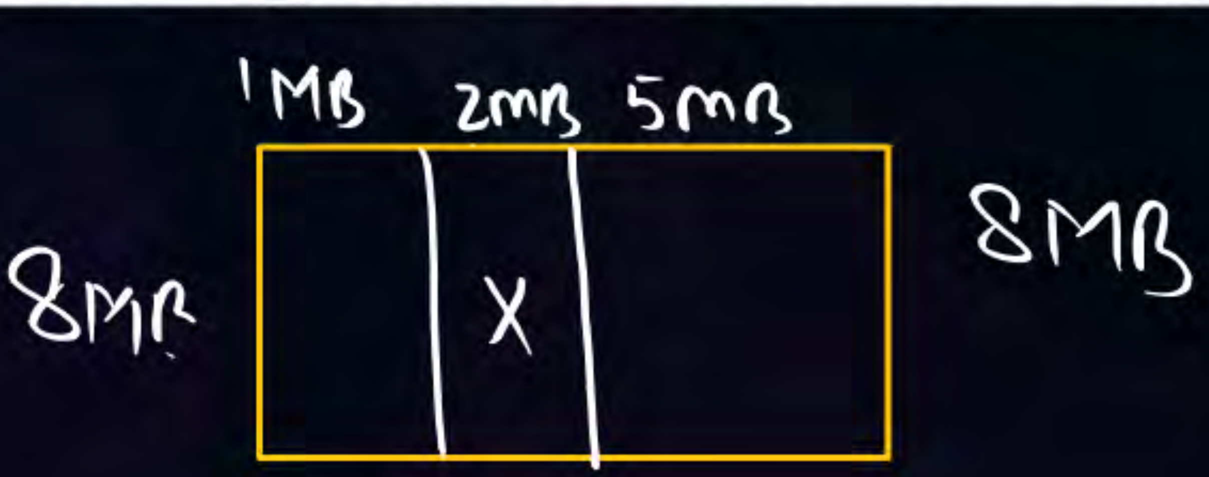
a. What was the maximum size of the swapped out process? 8MB

b. What was the size of the free block just before it was partitioned by X? 8MB

c. A new 3-Mbyte allocation request must be satisfied next. Indicate the intervals of memory where a partition will be created for the new process under the following four placement algorithms: best-fit, first-fit, next-fit, worst-fit. For each algorithm, draw a horizontal segment under the memory strip and label it clearly.

(i) 6

(ii) 9



a) F.F : 4M

b) B.F : 3M

c) N.F : 5M

d) W.F : 8M

How many max consecutive requests of size 3m can be satisfied with & without Compaction

Topic : Paging

- 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
- 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

I. Org. of L.A.S

L.A/V.A

$$\begin{array}{|c|c|}\hline P & d \\ \hline \end{array}$$

$$\xleftrightarrow{\hspace{3em}}$$

L.A format

1) $N_{pages} = \frac{L.A.S}{P.S}$

$$\boxed{L.A.S = N_{pages} * P.S}$$

2) $P = \lceil \log_2 N_{pages} \rceil$ bits ; $N_{pages} = 2^P$

3) Page offset $(d) = \lceil \log_2 PS \rceil$ bits

$P.S = 2^d$

II. orgnz. of P.A.S

1) $M_{frames} = \frac{P.A.S}{F.S}$ $(P.S = F.S)$

2) $f = \log_2 M_{frames}$ bits

$M = 2^f$

3) Frame offset = Page offset = d

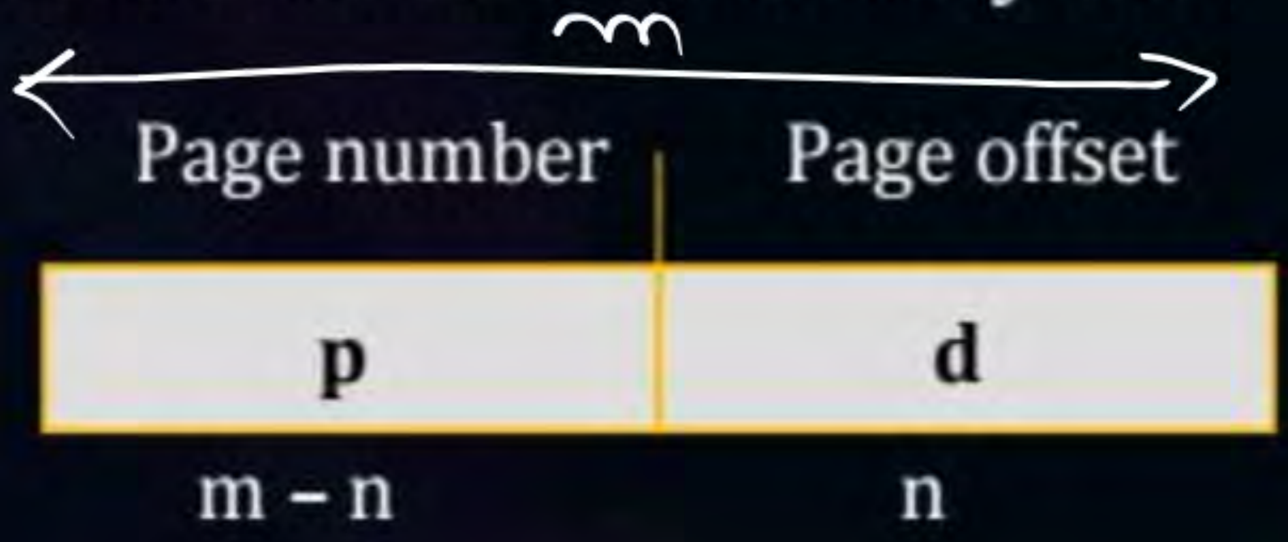
4)

$$\begin{array}{|c|c|}\hline f & d \\ \hline \end{array}$$

P.A

Topic : Address Translation Scheme

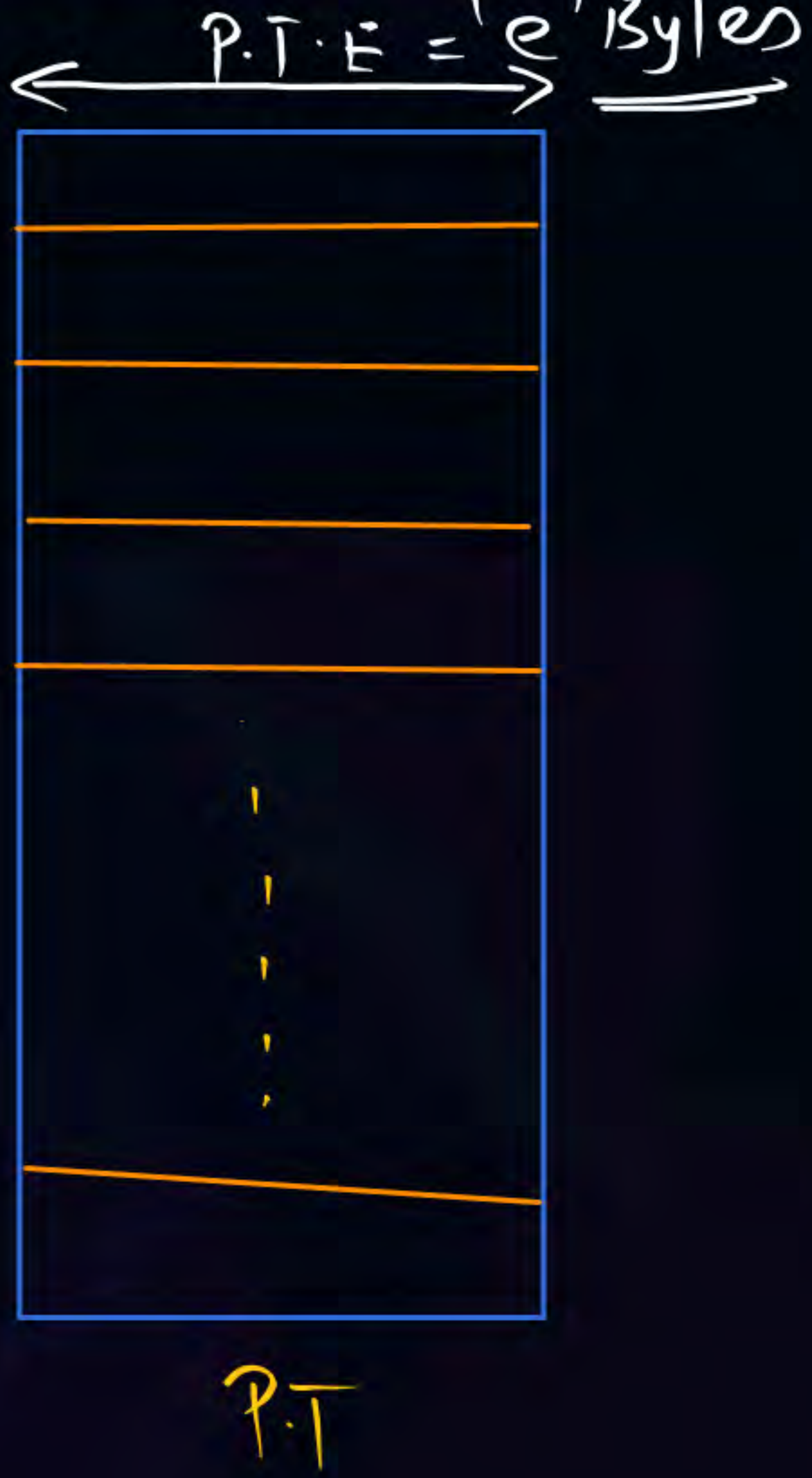
- Address generated by CPU is divided into:
  - **Page number (p)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



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

III. Orgnz. of P.T.

$$P.T.S = N * e$$



→ No. of entries in P.T = No. of Pages in L.A.S

→ P.T. Entries Contain frame No.

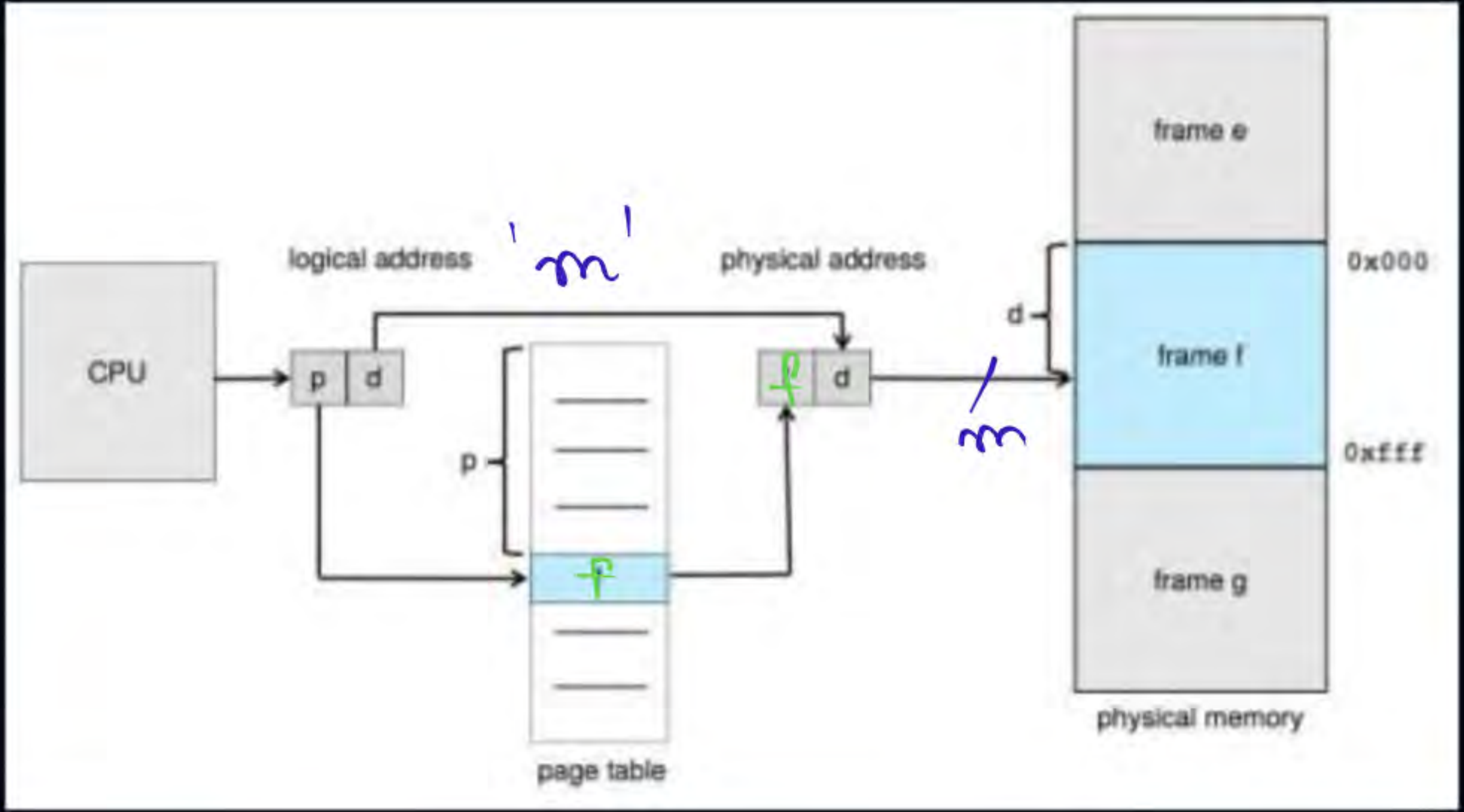
→ Each process has its own P.T

→ P.T's are Stored in M.M

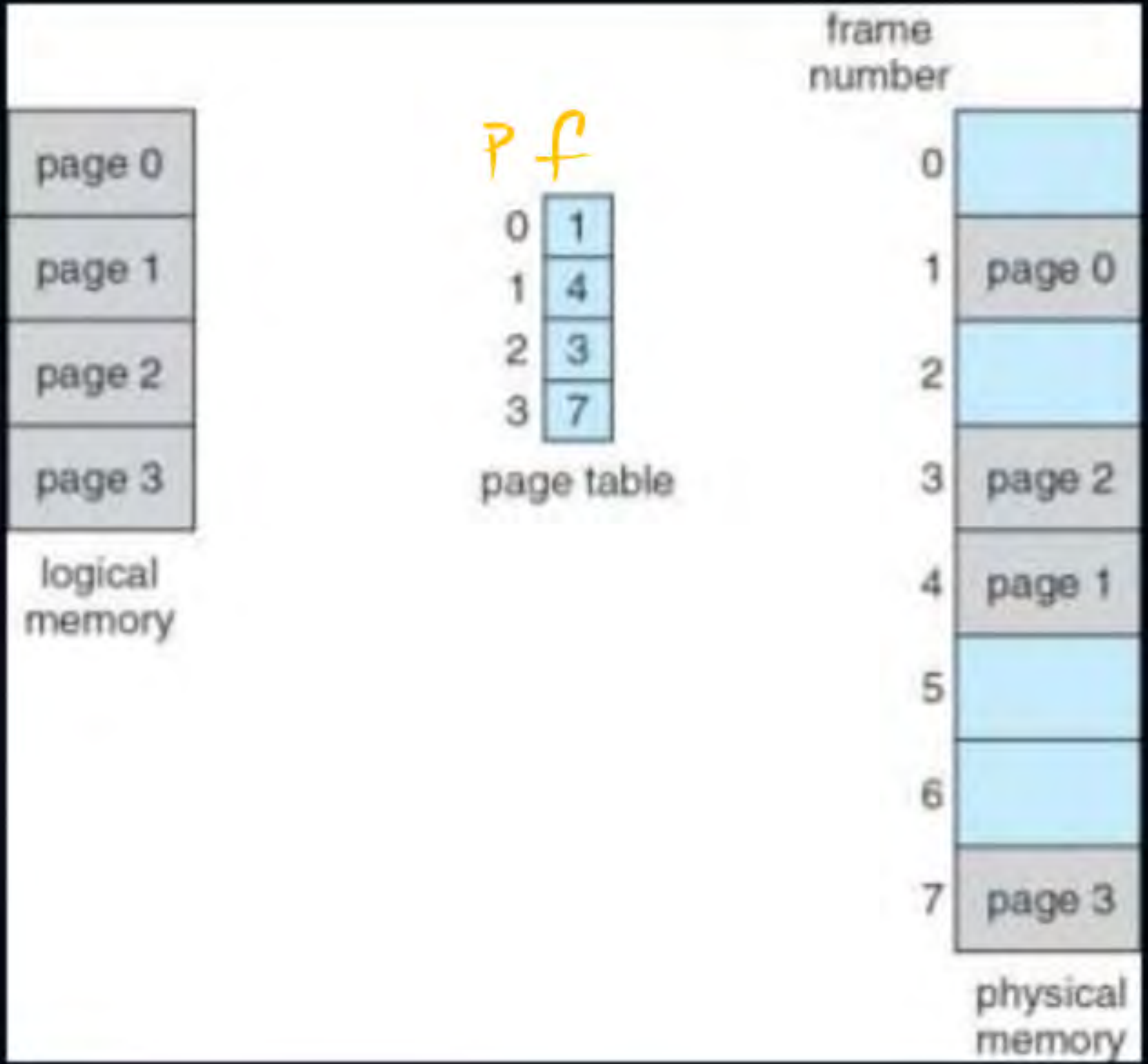
Topic : Paging Hardware

$M.M.A.T = 'm'$

$E.M.A.T_{S.P} = 2.m \; (\underset{P.T}{m} + \underset{\text{Content I/D access}}{m})$



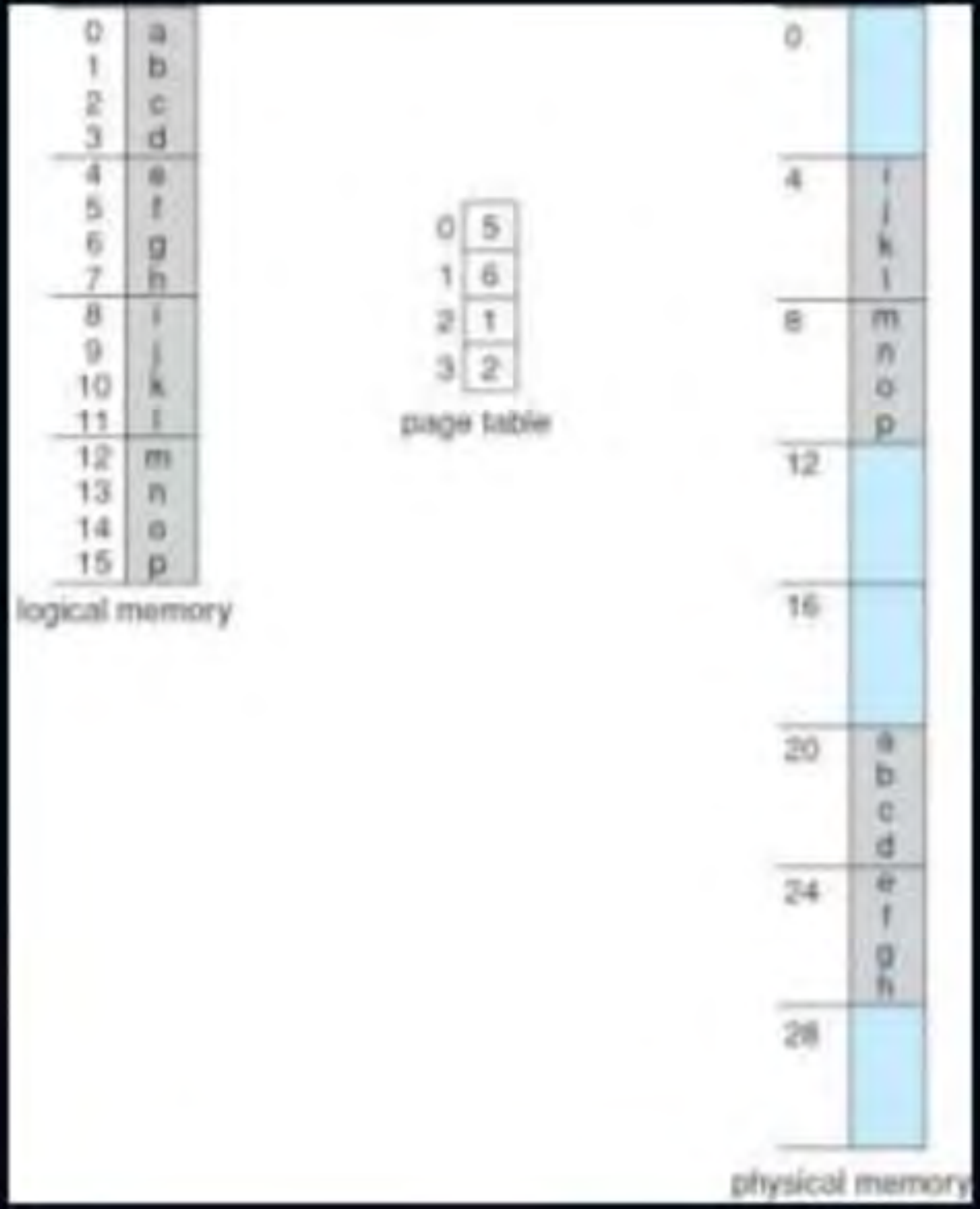
Topic : Paging Model of Logical and Physical Memory



p	f
0	1
1	4
2	3
3	7

Topic : Paging Example

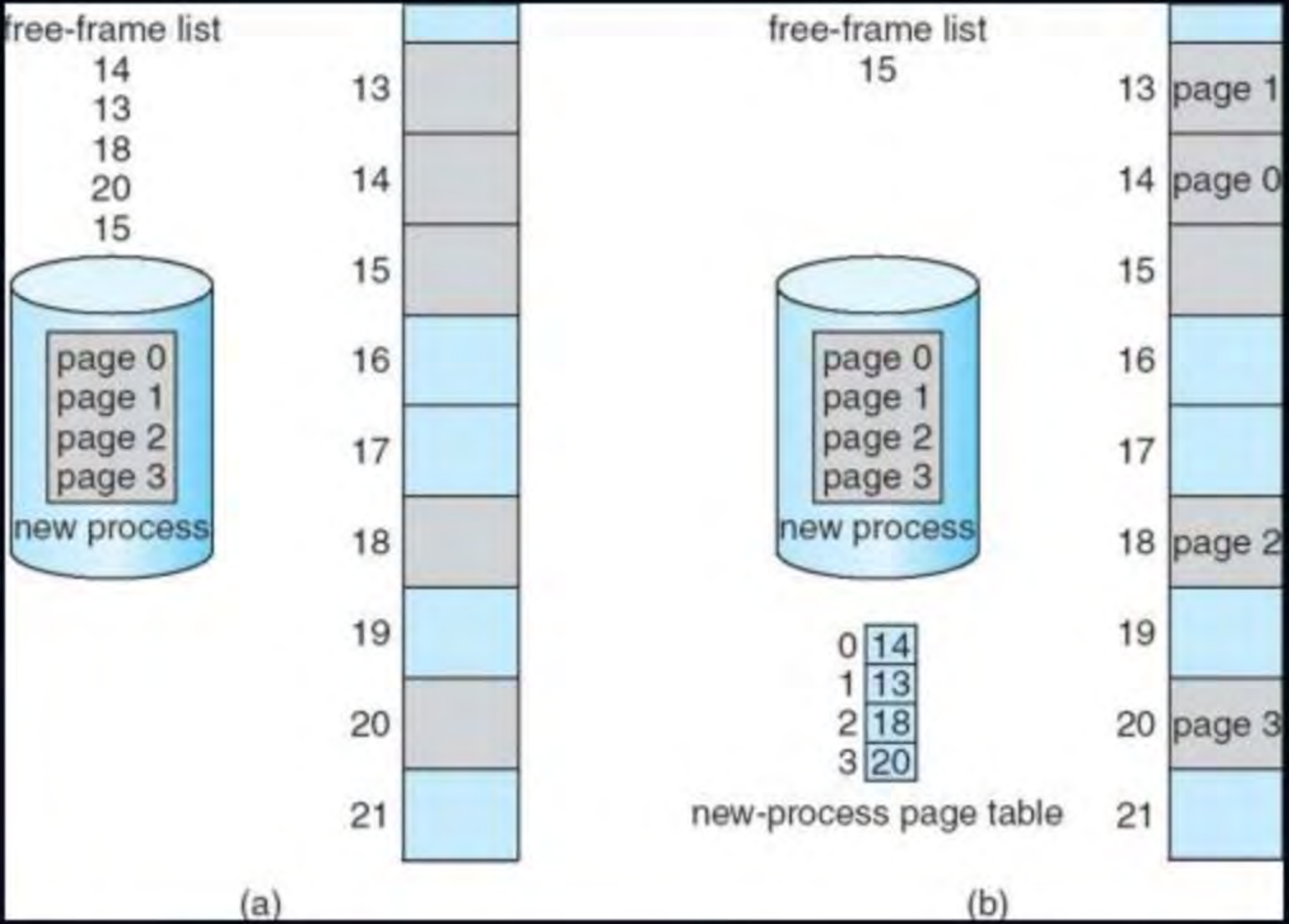
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



Topic : 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
  - Solaris supports two page sizes – 8 KB and 4 MB

Topic : Free Frames

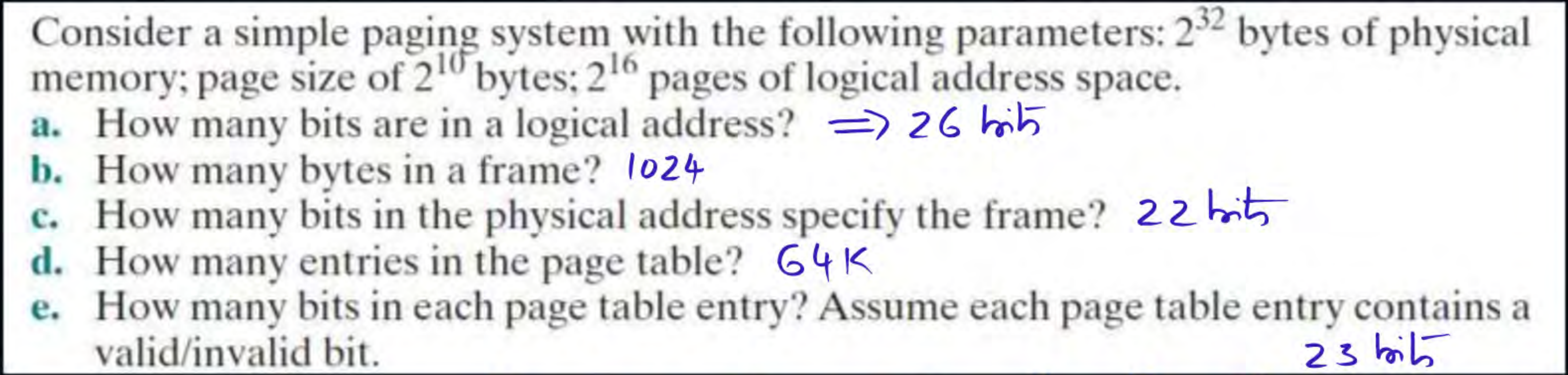


Before allocation

After allocation

Topic : Implementation of page Table

- Page table is kept in main memory P.C.B
  - **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 **translation look-aside buffers (TLBs)** (also called **associative memory**).



Consider a simple paging system with the following parameters: 2^{32} bytes of physical memory; page size of 2^{10} bytes; 2^{16} pages of logical address space.

a. How many bits are in a logical address? ⇒ 26 bits

b. How many bytes in a frame? 1024

c. How many bits in the physical address specify the frame? 22 bits

d. How many entries in the page table? 64K

e. How many bits in each page table entry? Assume each page table entry contains a valid/invalid bit.

23 bits

a) $N = 2^{16}$; $P.S = 2^{10}$; $L.A.S = 2^{16} \times 2^{10} = 2^{26} = \underline{64MB}$ ✓

$L.A = 26$

c) $P.A.S = 2^{32}$

$M = 2^{32}/2^{10} = 2^{22}$, $f = 22$ bits

d) $N = 2^{16} = 64K$

e) $P.T.E = \boxed{f + V|I}$: $22 + 1 = \underline{\underline{23 \text{ bits}}}$.

PYQ

Q) L·A = 32 bits; P·S = 4KB; P·A·S = 64 MB;

What is the approximate P.T. Size in Bytes ?

a) 16 MB b) 8MB c) 2MB d) 24 MB

$\boxed{P.T.S = N * e}$ $e \geq 1B$

$e \sim \text{'}\underline{f}\text{'} + \text{Add. info}$

$N = \frac{2^{32}}{2^{12}} = 2^{20} = \underline{\underline{1M}}$

$M = 2^{26}/2^{12} = 2^{\textcircled{14}}$ f

$\boxed{f \mid Add}$

$e = 14 + 2 = 16 \text{ bits}$
$= \underline{\underline{2B}}$
(approx)

$P.T.S = 1M * 2B$
$= \underline{\underline{2MB}}$ ✓

$N = 2^{20} = 1M$

$f = 14 \text{ bits}$

$P.T.S = \frac{1M \times 14 \text{ bits}}{8}$

$= 2^{20-3} * 14 B$

$= 128 \times 14 KB$

$= 1792 KB = \underline{1.75MB}$

Topic : Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – 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

Topic : Hardware

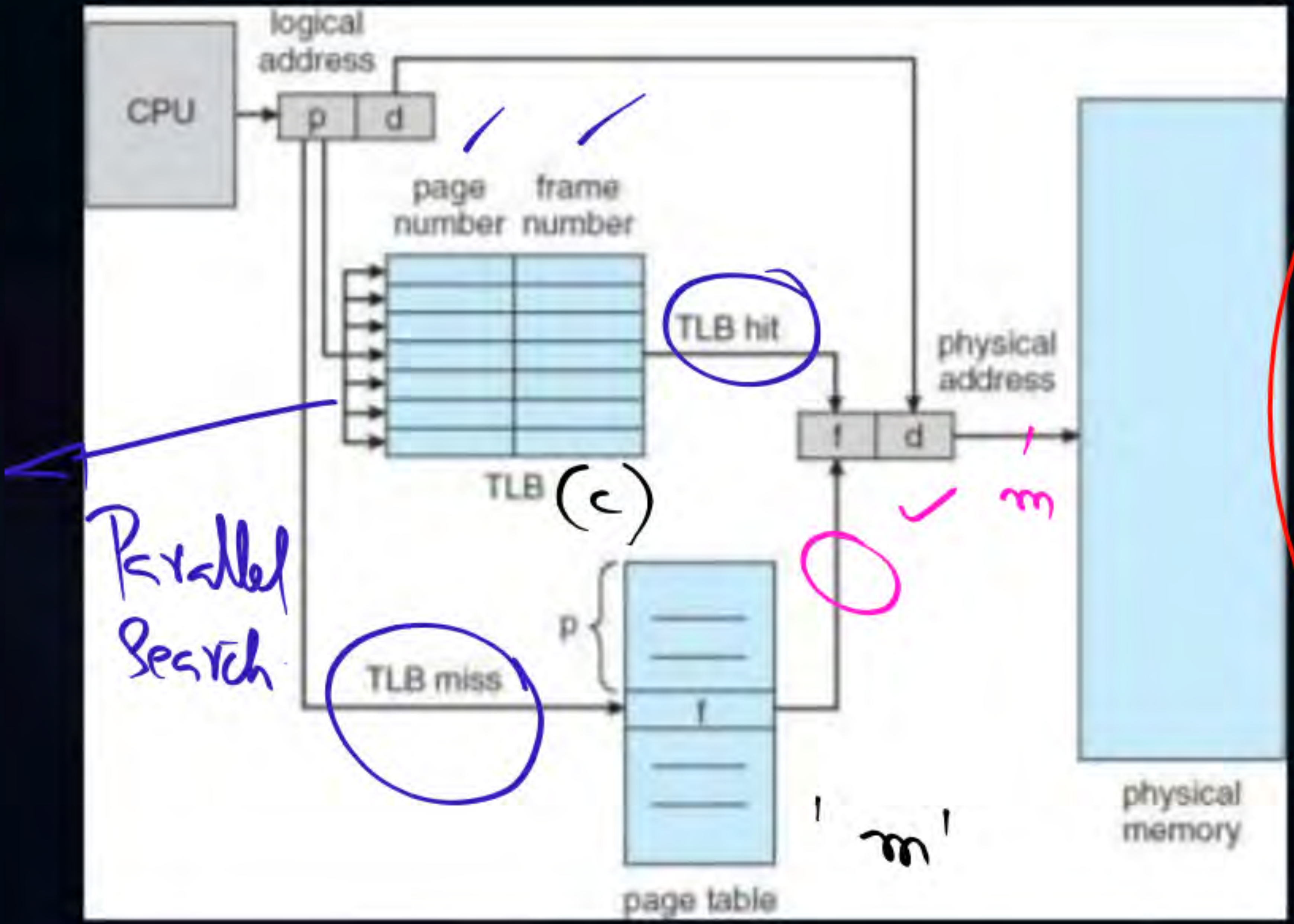
- Associative memory – parallel search

Page #	Frame #

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

Topic : Paging Hardware With TLB

L. Addr. Cache
+
P. Addr. Cache



$$EMAT = x(c+m) + (1-x)(c+2m)$$

TLB + SP

Topic : Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.

m c = 0

- Suppose that $\underline{10}$ nanoseconds to access memory.

  If we find the desired page in TLB then a mapped-memory access take 10 ns

  Otherwise we need two memory access so it is $\underline{20 \text{ ns}}$
- **Effective Access Time (EAT)**

  $$\text{EAT} = \underline{0.80} \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

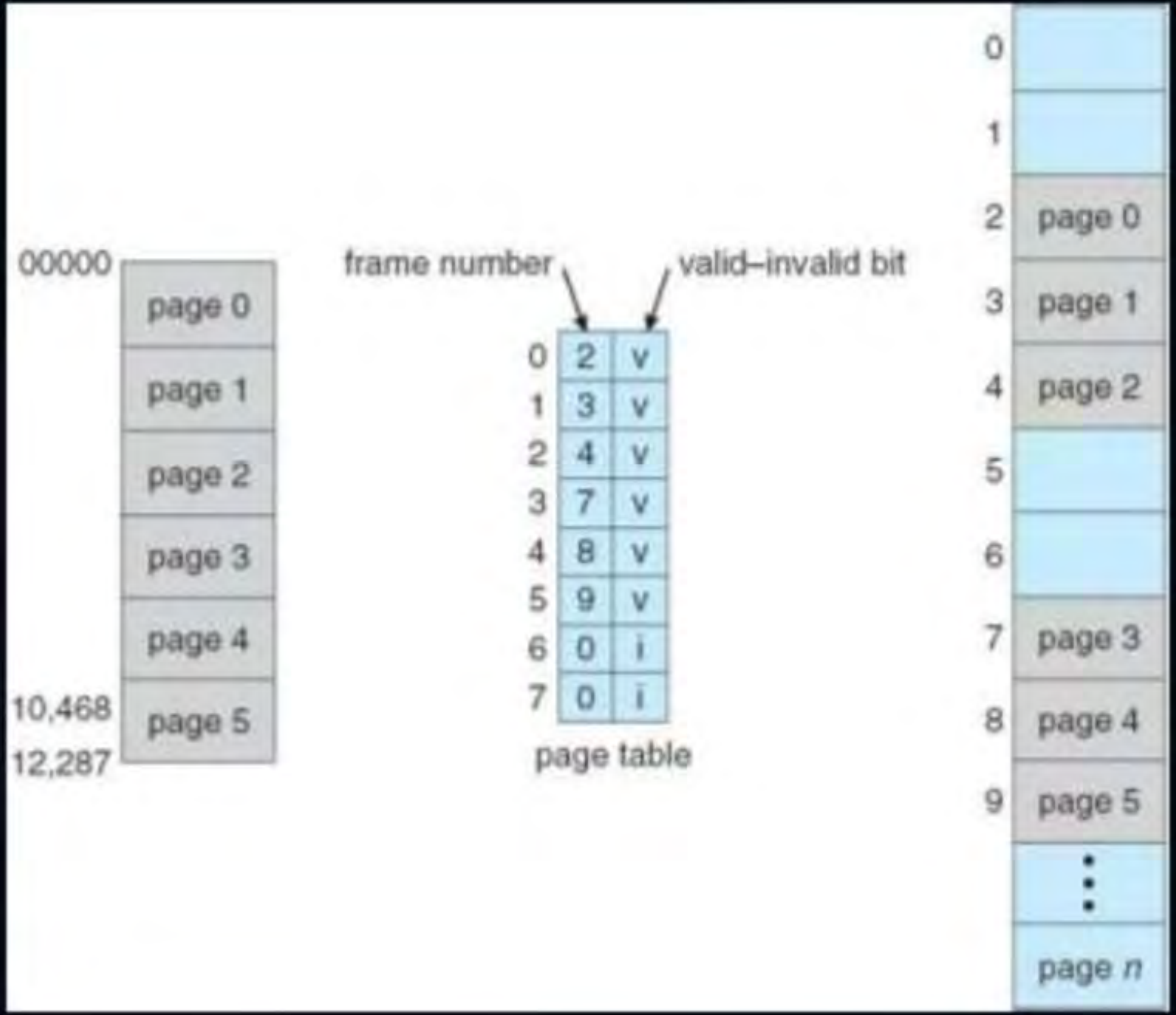
  $$\text{EAT} = \underline{0.99} \times 10 + 0.01 \times 20 = \boxed{10.1\text{ns}}$$

  implying only 1% slowdown in access time.

Topic : 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

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

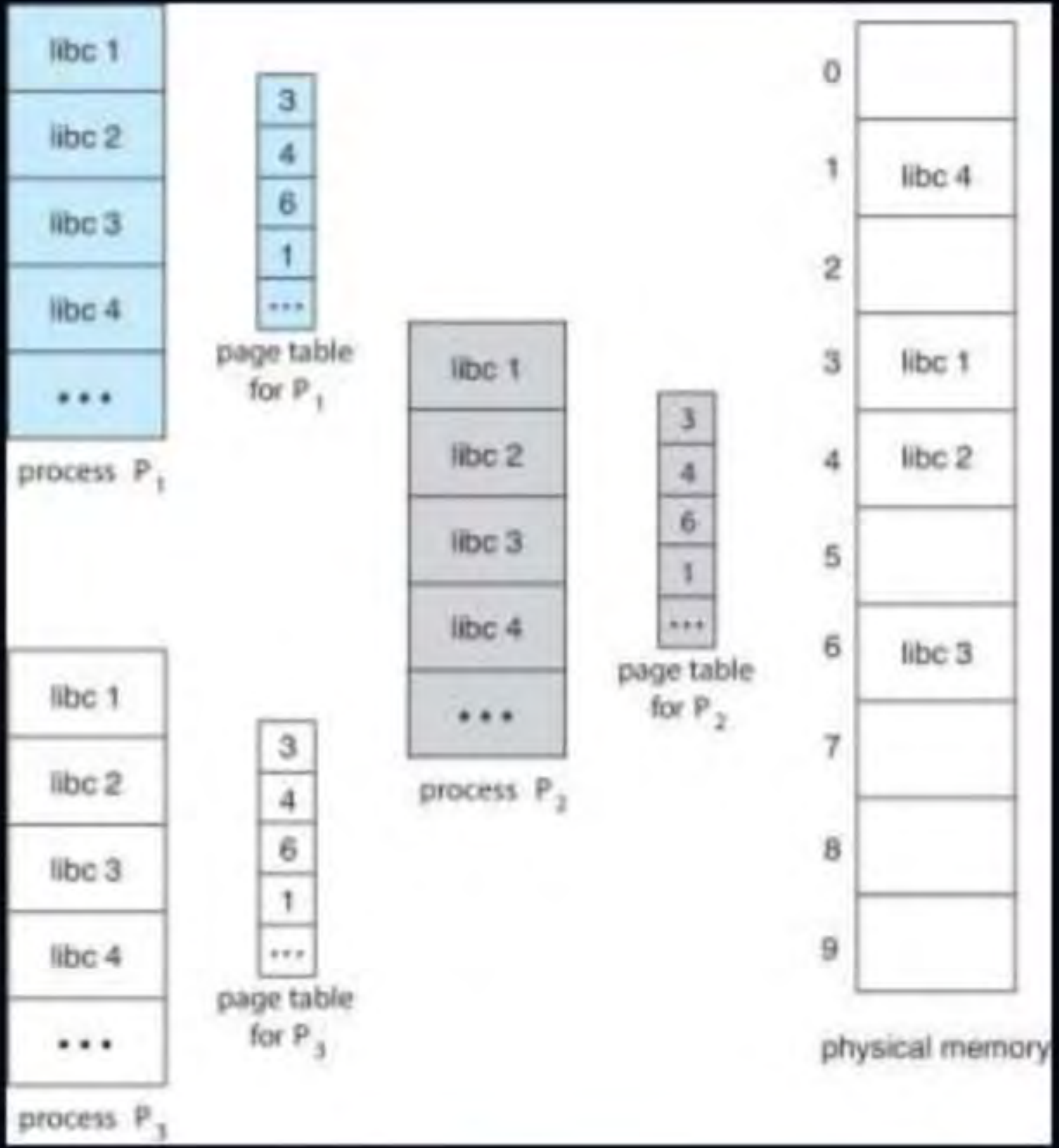


	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Topic : 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

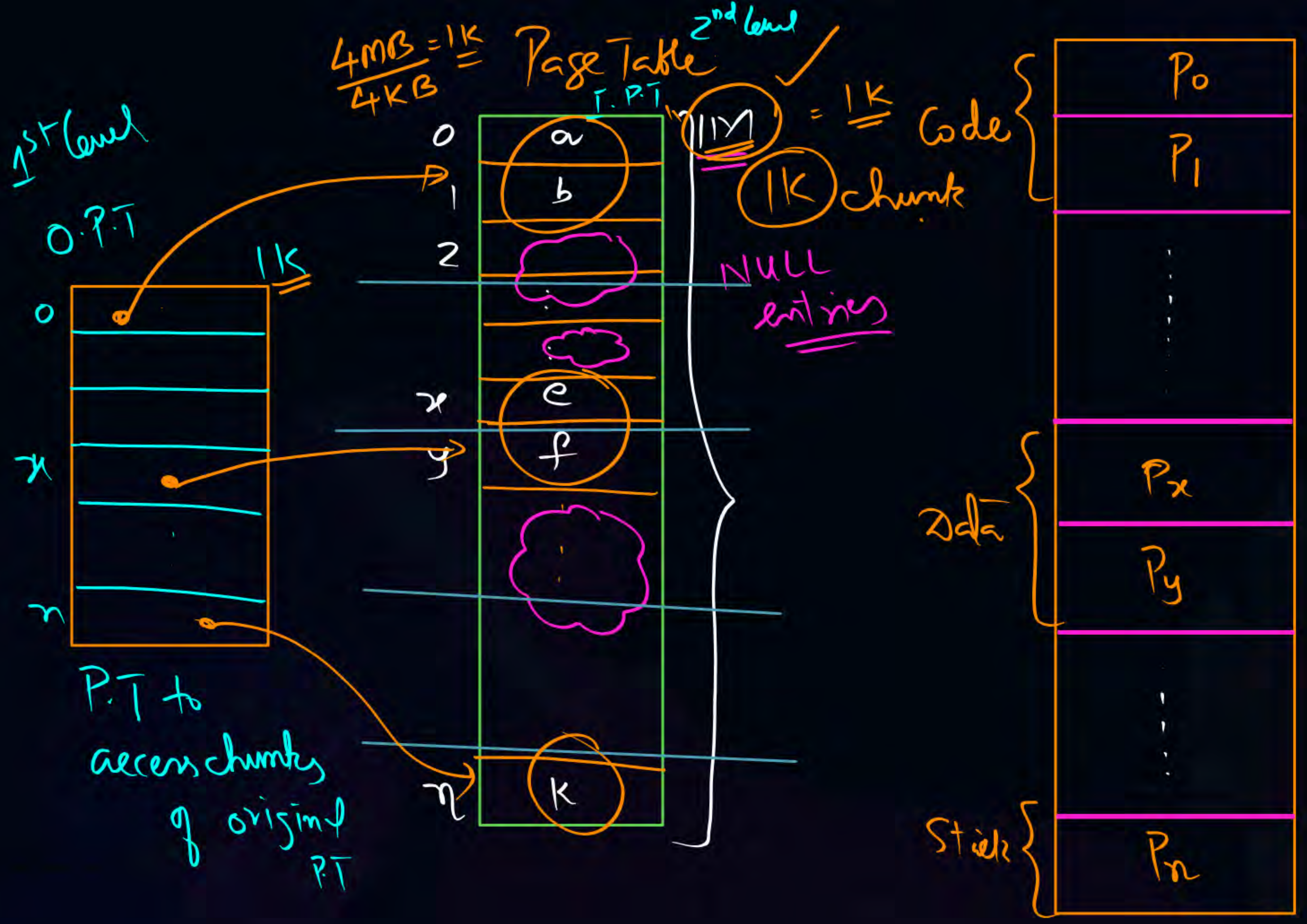
Topic : Shared Pages Example



Topic : Structure of the page Table

Multi-Level Paging (MLP)

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12}) $N = 2^{32}/2^{12} = 1M$
  - If each entry is 4 bytes ➔ each process 4 MB of physical address space for the page table alone
    - ✓ Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - ✓ Hierarchical Paging ✓ — M.L.P
    - ✓ Hashed Page Tables
    - ✓ Inverted Page Tables



L.A.S = 2^{32} = 4GB

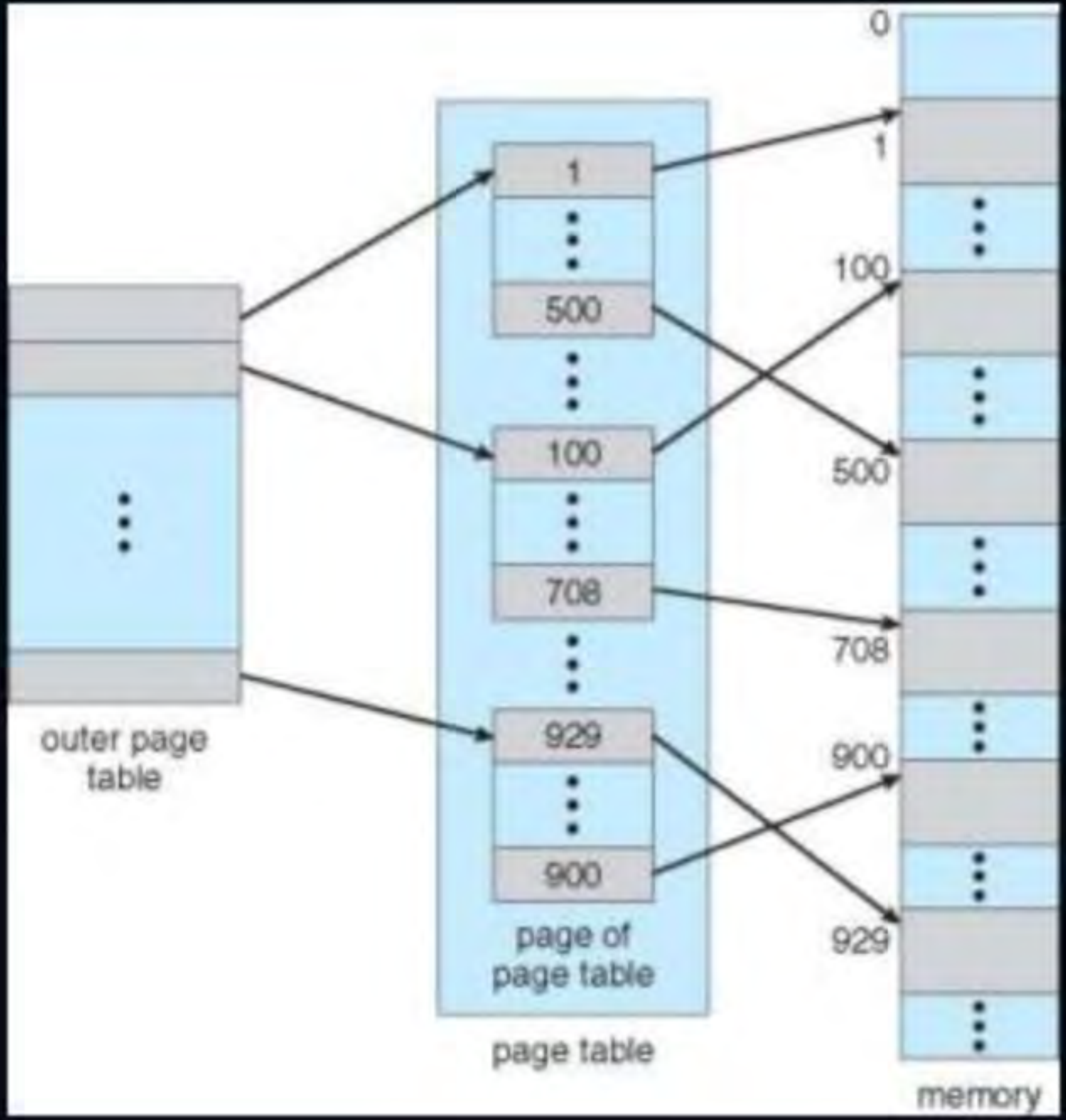
P.S = 4KB

N = 2^{20} = 1M

Typical Process
↳ 5 Pages
  ↳ 2 Pg Code
  ↳ 2 Pg Data
  ↳ 1 Stack

Topic : Hierarchical Page Tables

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



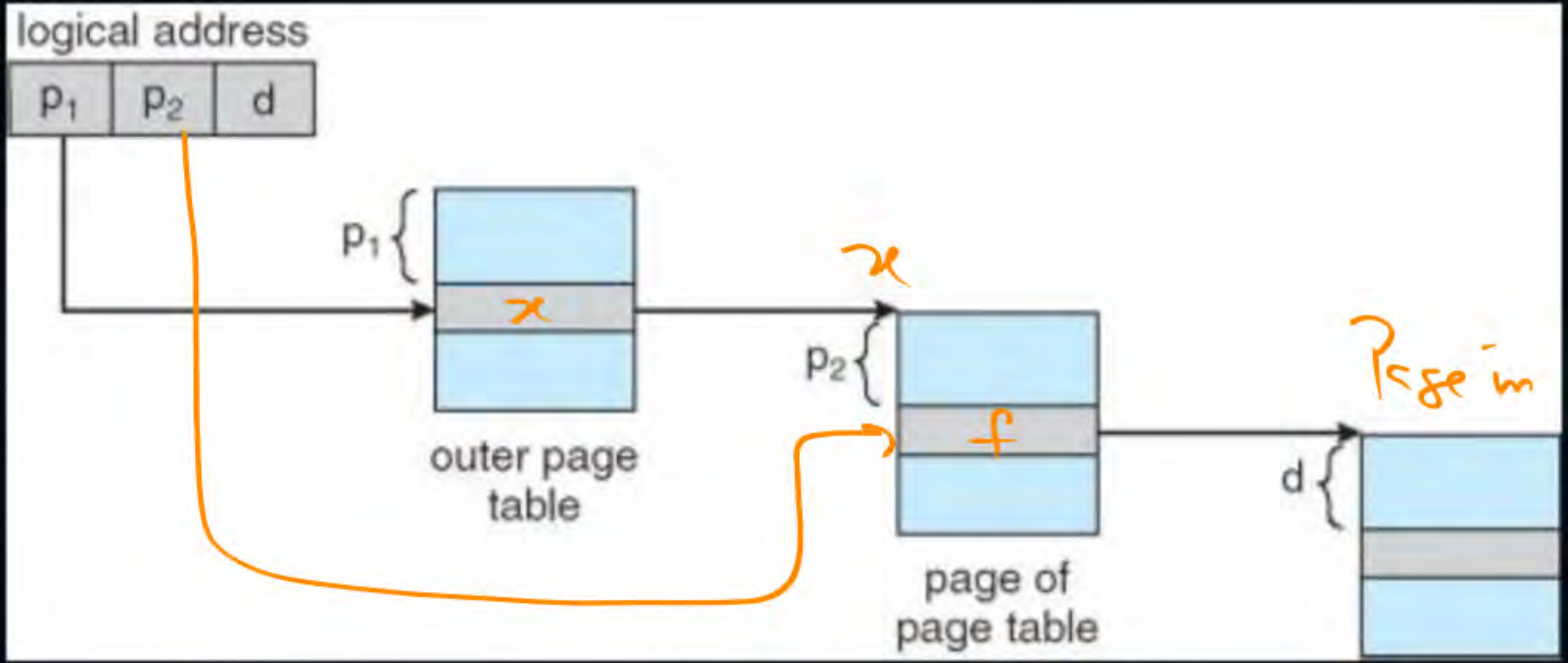
Topic : Two-Level Paging Example

- 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_1	p_2	d
10	10	12

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Topic : Translation Scheme

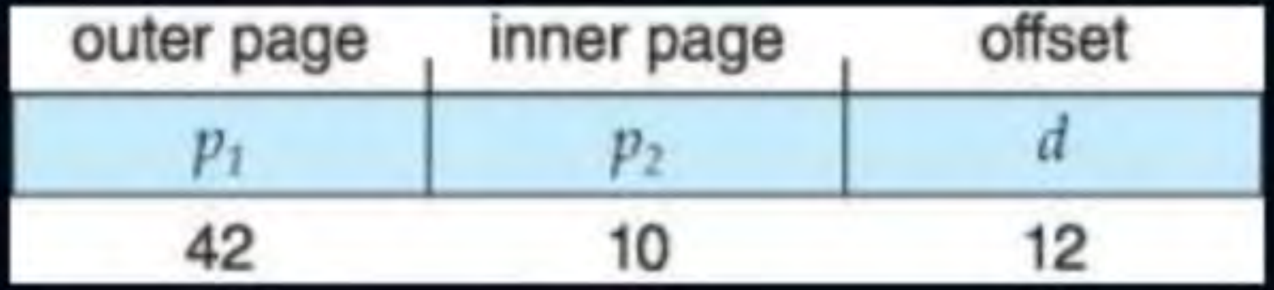


Page in P.A.S

Chunk of P.T

Topic : 64-bit Logical Address Space

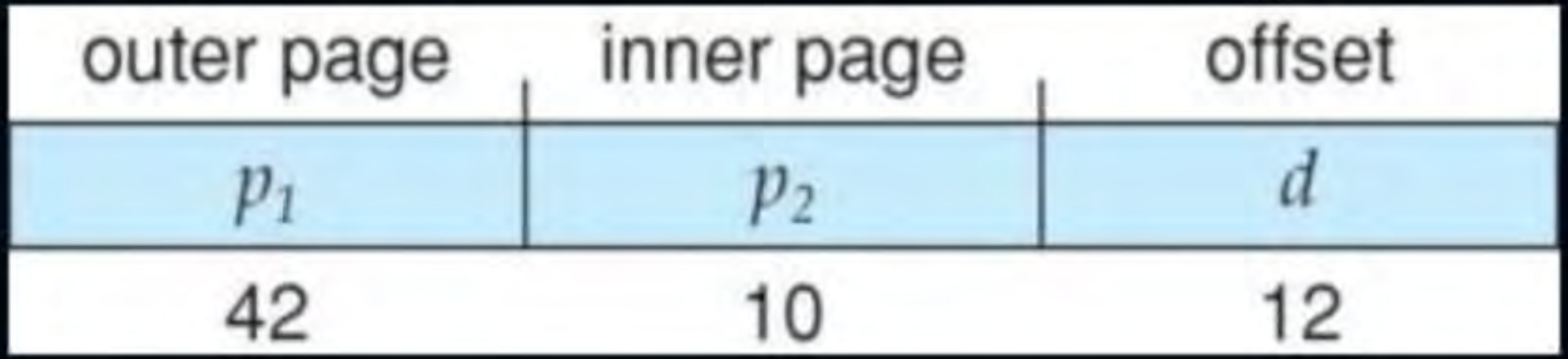
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



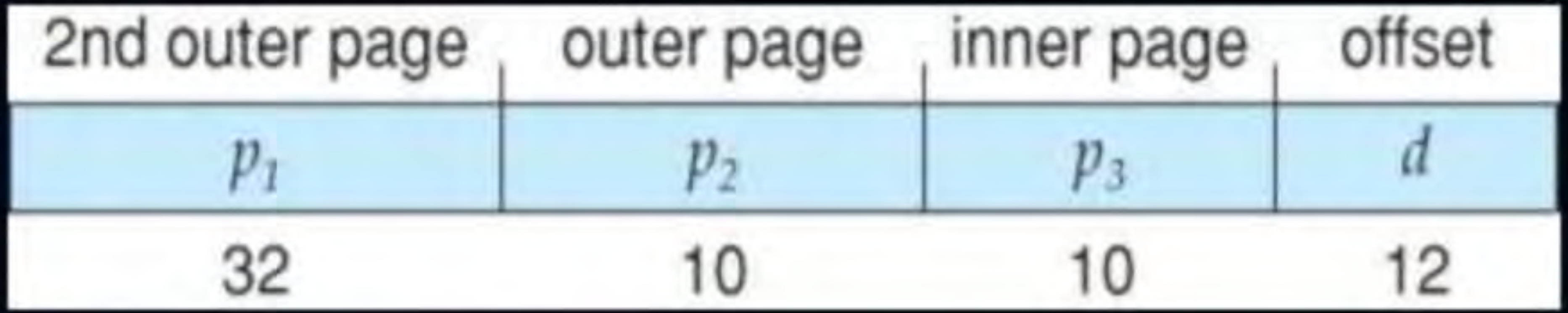
outer page	inner page	offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes in size
    - ✓ And possibly 4 memory access to get to one physical memory location

Topic : Three-level Paging Scheme



outer page	inner page	offset
p_1	p_2	d
42	10	12

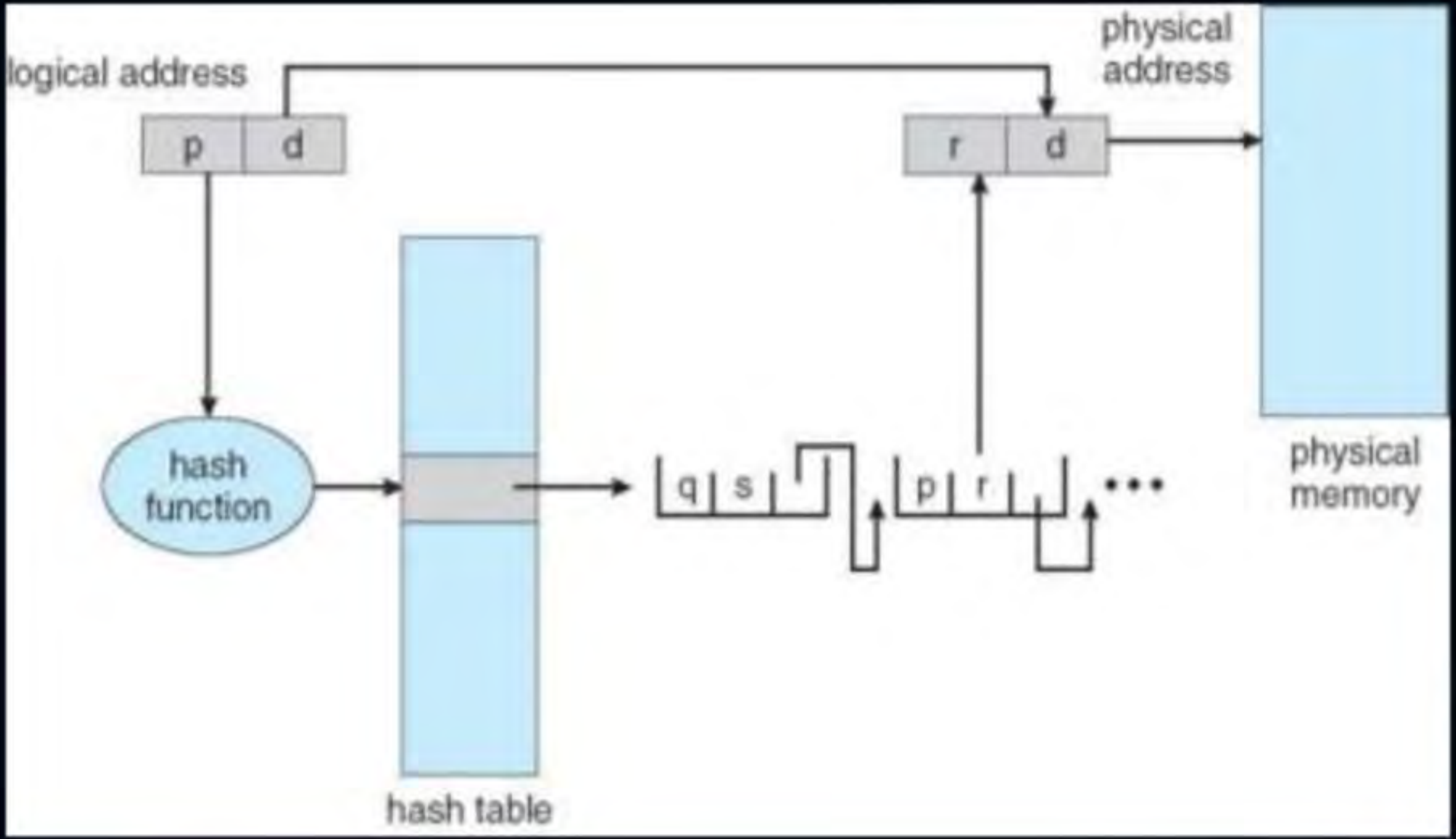


2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Topic : Hashed Page Table

- 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
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
    - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
    - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
    - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

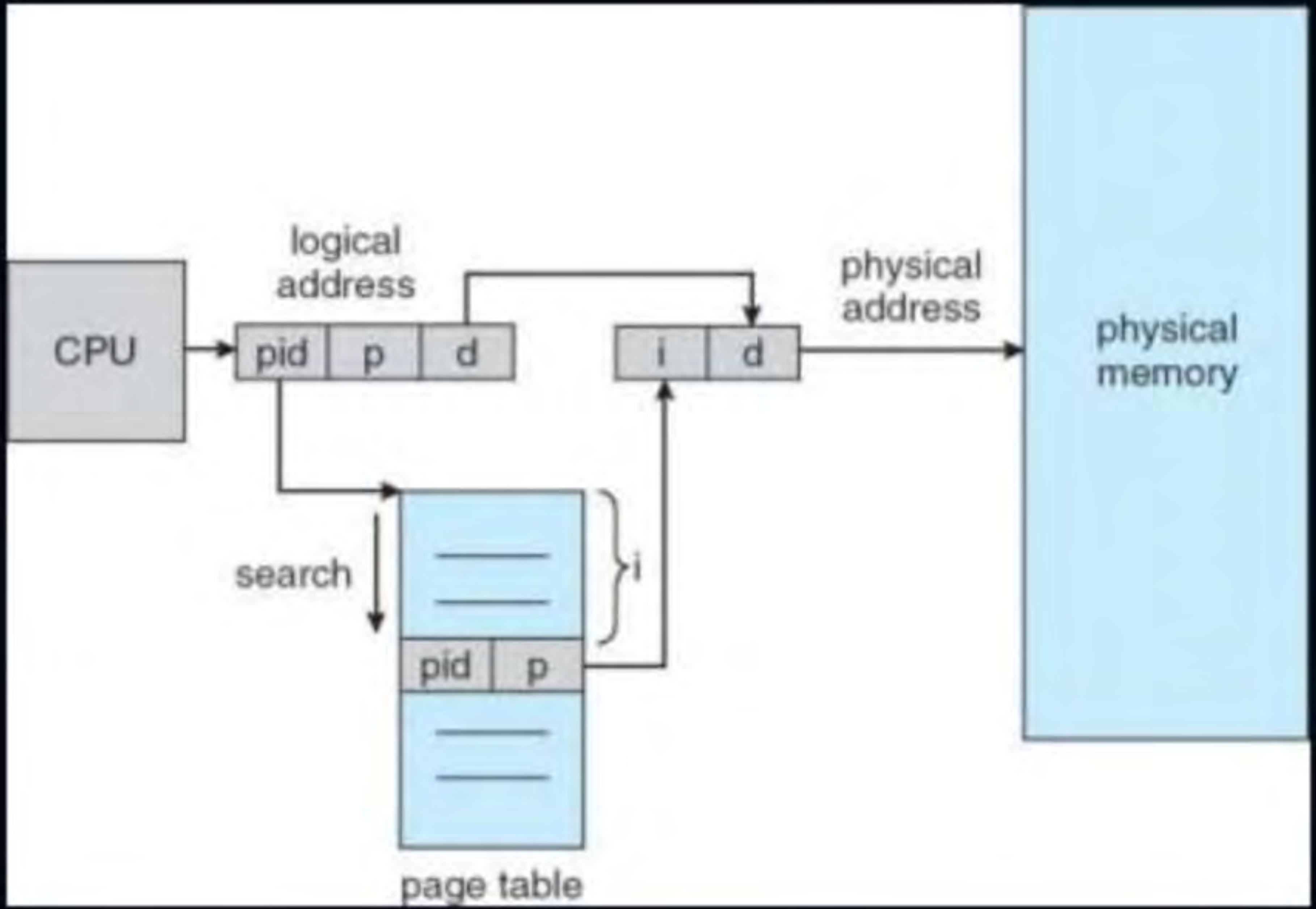
Topic : Hashed Page Table



Topic : Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Topic : Inverted Page Table Architecture



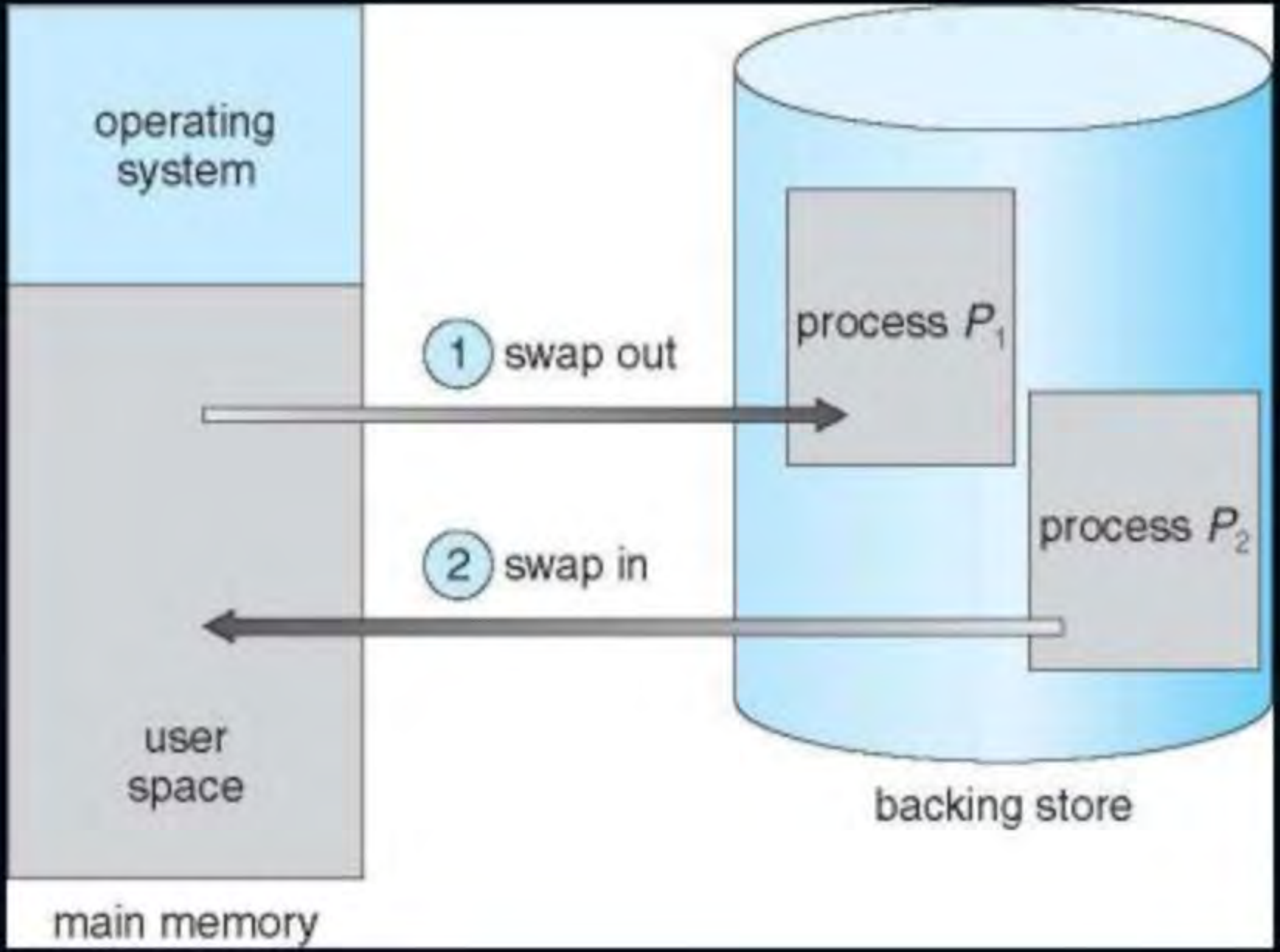
Topic : Swapping

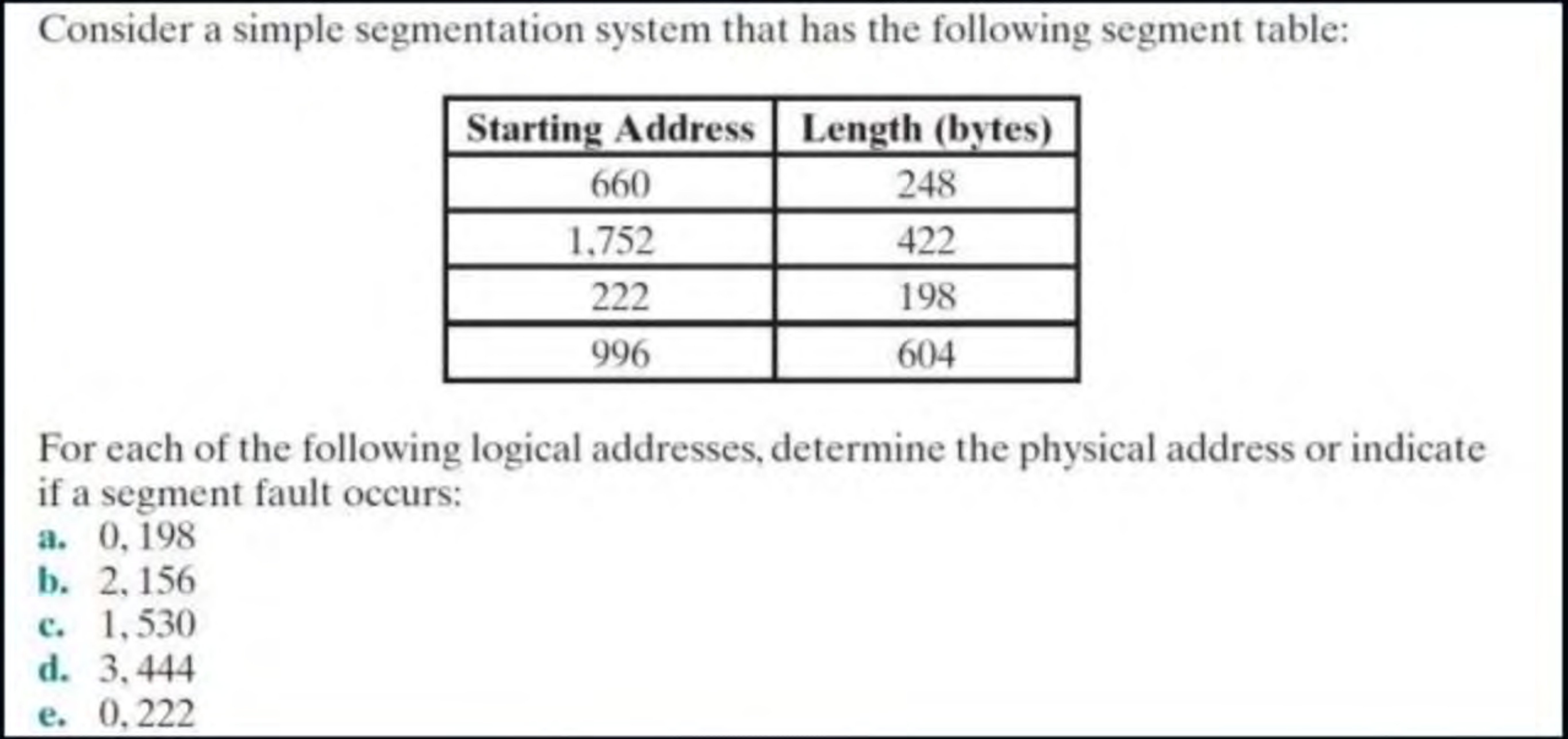
- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for 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
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Topic : Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Topic : Schematic View of Swapping





Consider a simple segmentation system that has the following segment table:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

For each of the following logical addresses, determine the physical address or indicate if a segment fault occurs:

a. 0, 198

b. 2, 156

c. 1, 530

d. 3, 444

e. 0, 222

2 mins Summary

Topic One Variable Partition

Topic Two Schematic View of Swapping

Topic Three Swapping

Topic Four

43

Topic Five

THANK - YOU