# Operating Systems
# CS2006

Lecture 14

**Memory Management-II**

17th April 2023

Dr. Rana Asif Rehman

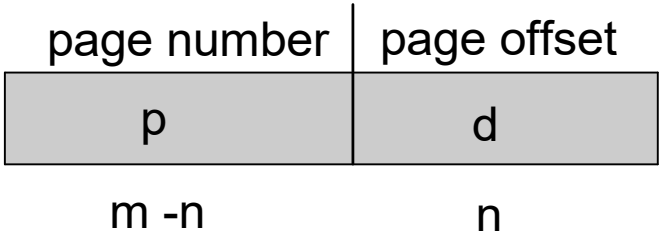
# Paging

# 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 1 GB per page
- 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

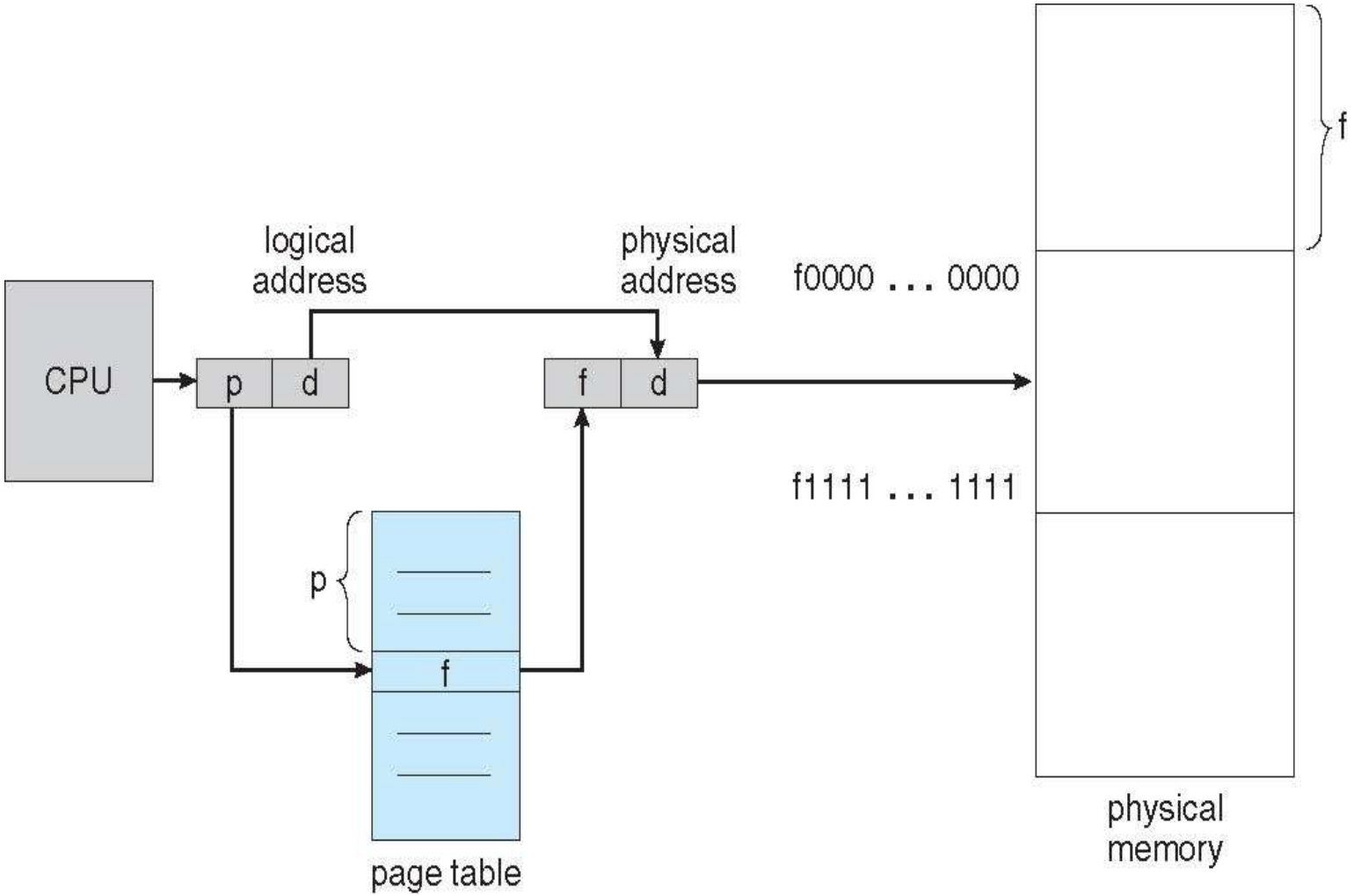
# Address Translation Scheme

- 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
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

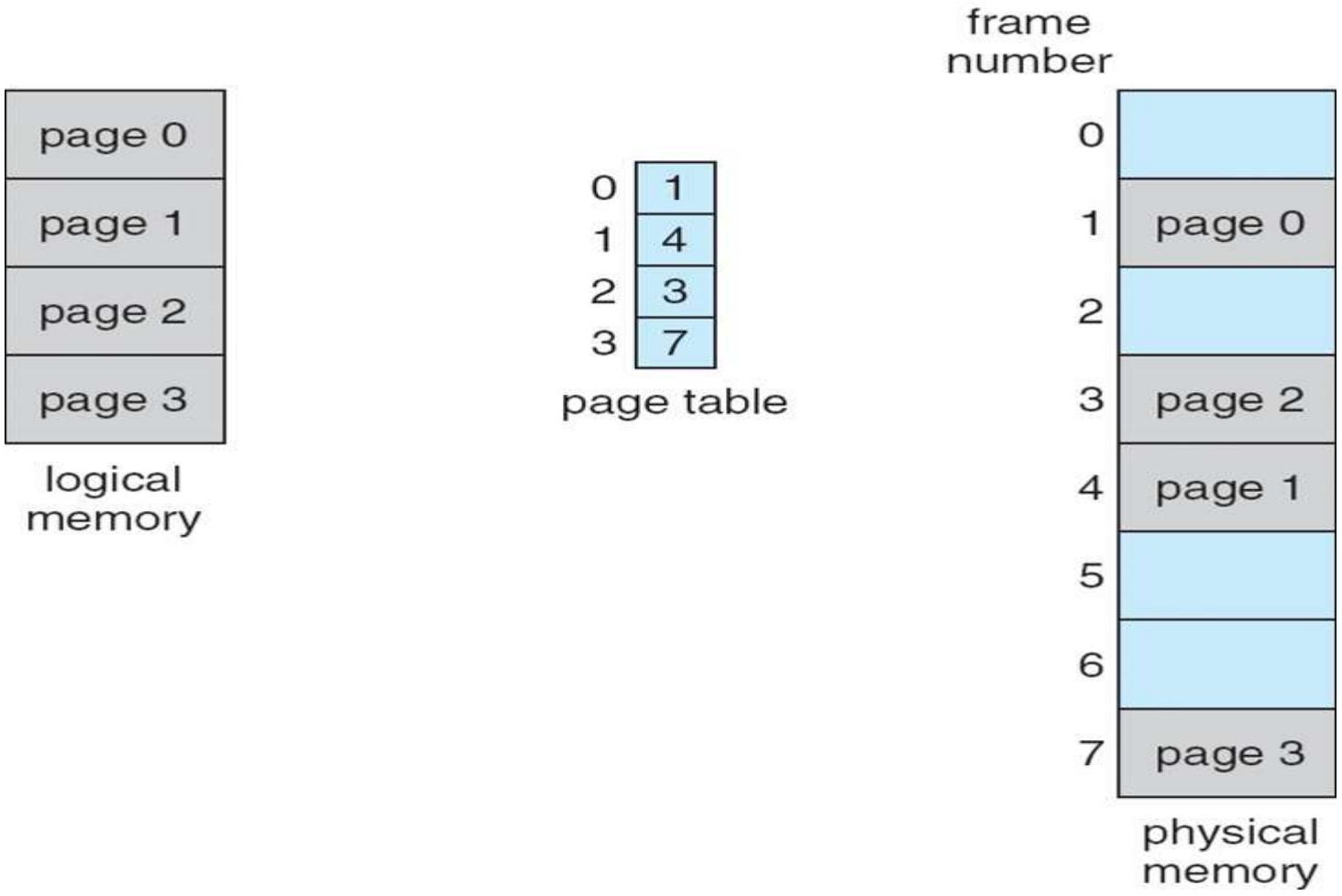



  - For given logical address space $2^m$ and page size $2^n$

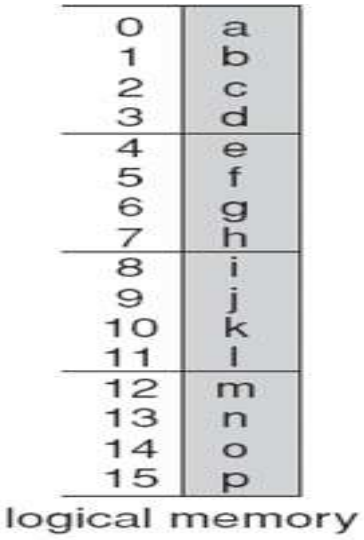
# Paging Hardware

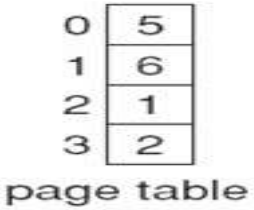


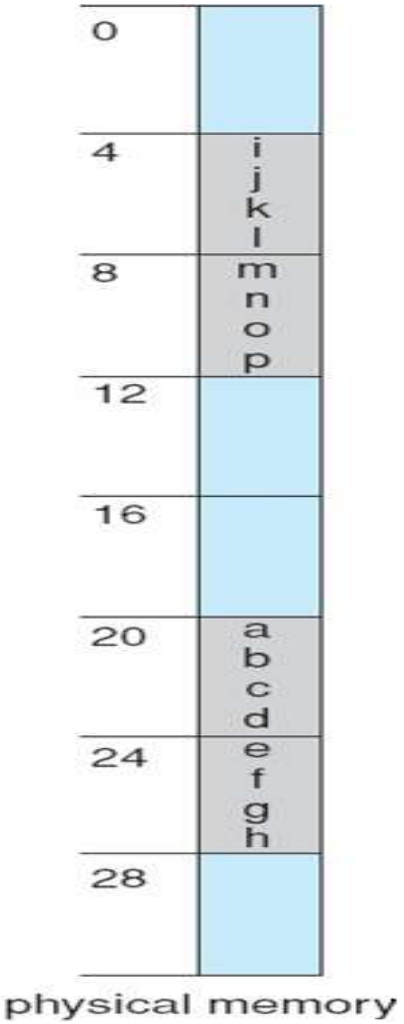
# Paging Model of Logical and Physical Memory



# Paging Example



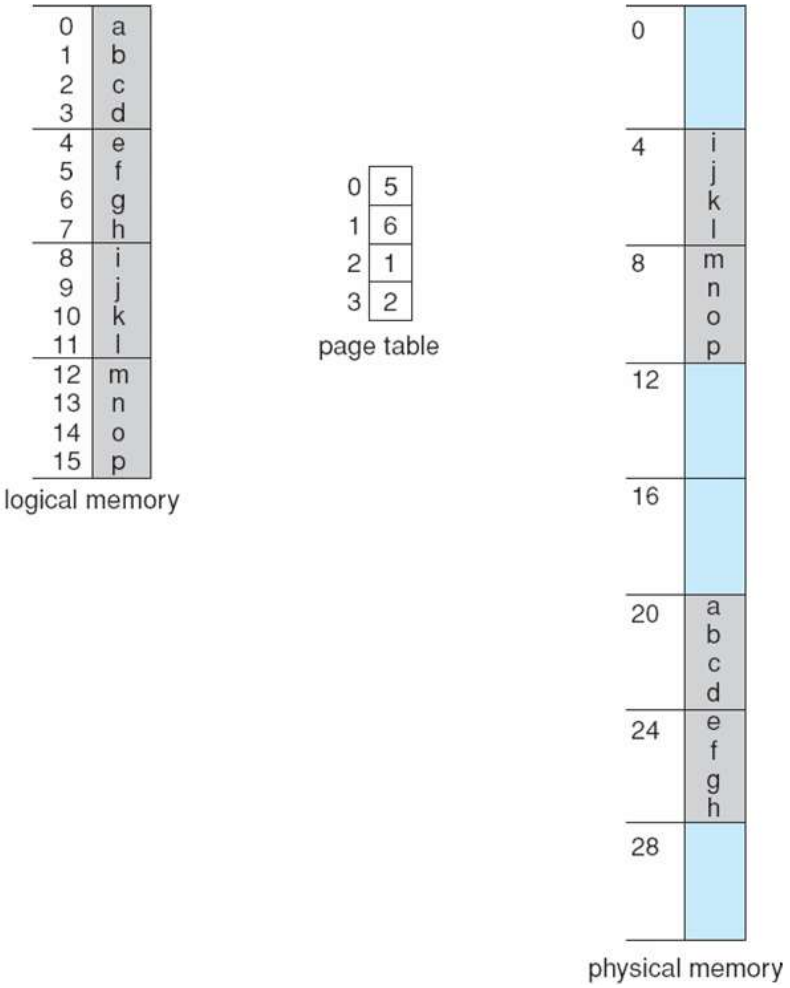

*n*=2 and *m*=4  32-byte memory and 4-byte pages

# Paging Example

32-byte memory and 4-byte pages

(*n*=2 and *m*=4)

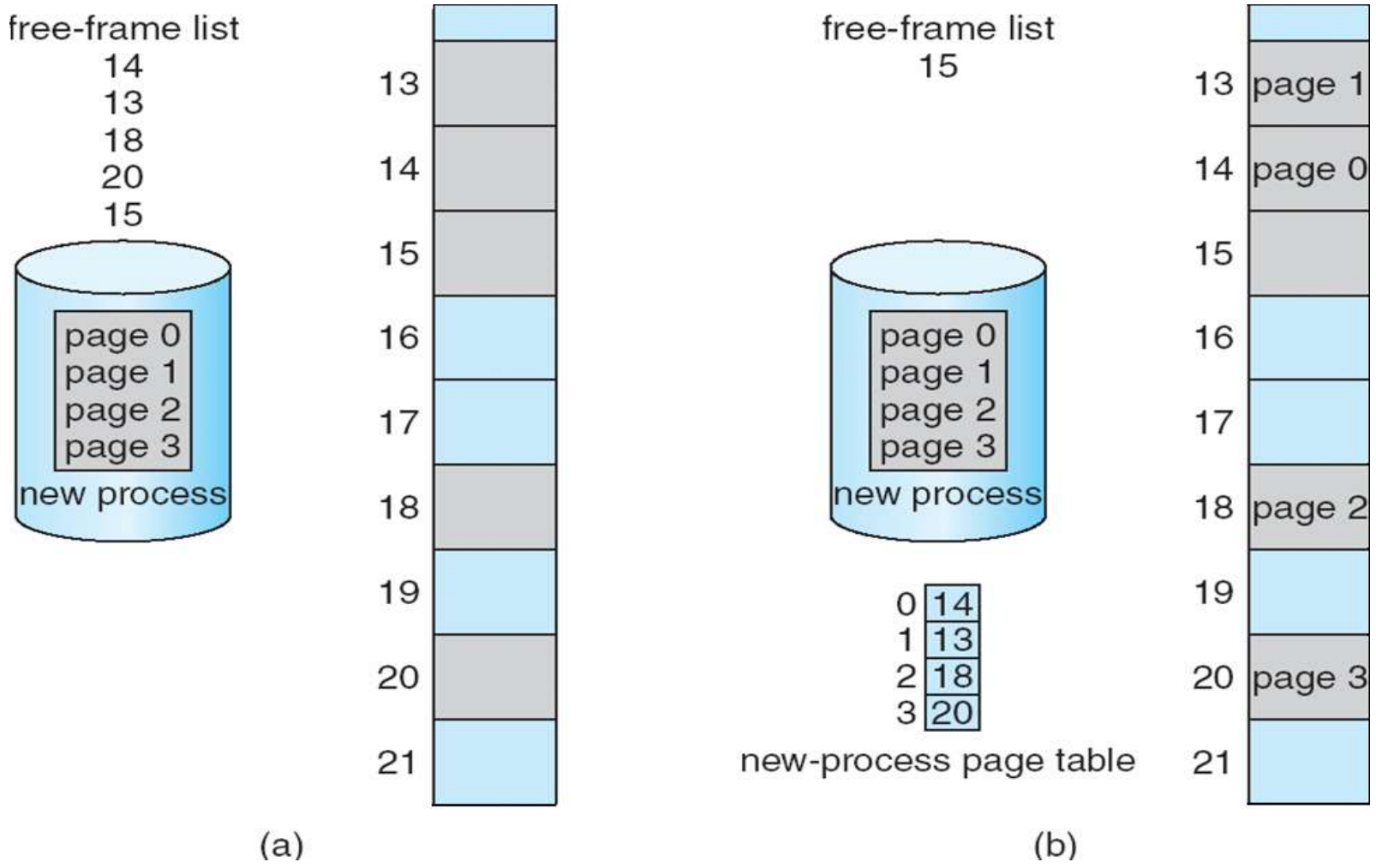
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
- Logical address 0 is page 0, offset 0.
- Page 0 is in frame 5.
- Logical address 0 maps to physical address 20 [= (5 × 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.
- Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].
- Logical address 13 maps to physical address 9.



# Paging (Cont.)

- 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
- Process view and physical memory now very different
- By implementation process can only access its own memory

# Free Frames




Before allocation

After allocation

# 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 look-aside buffers** (**TLBs**)

# Implementation of Page Table (Cont.)

- 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

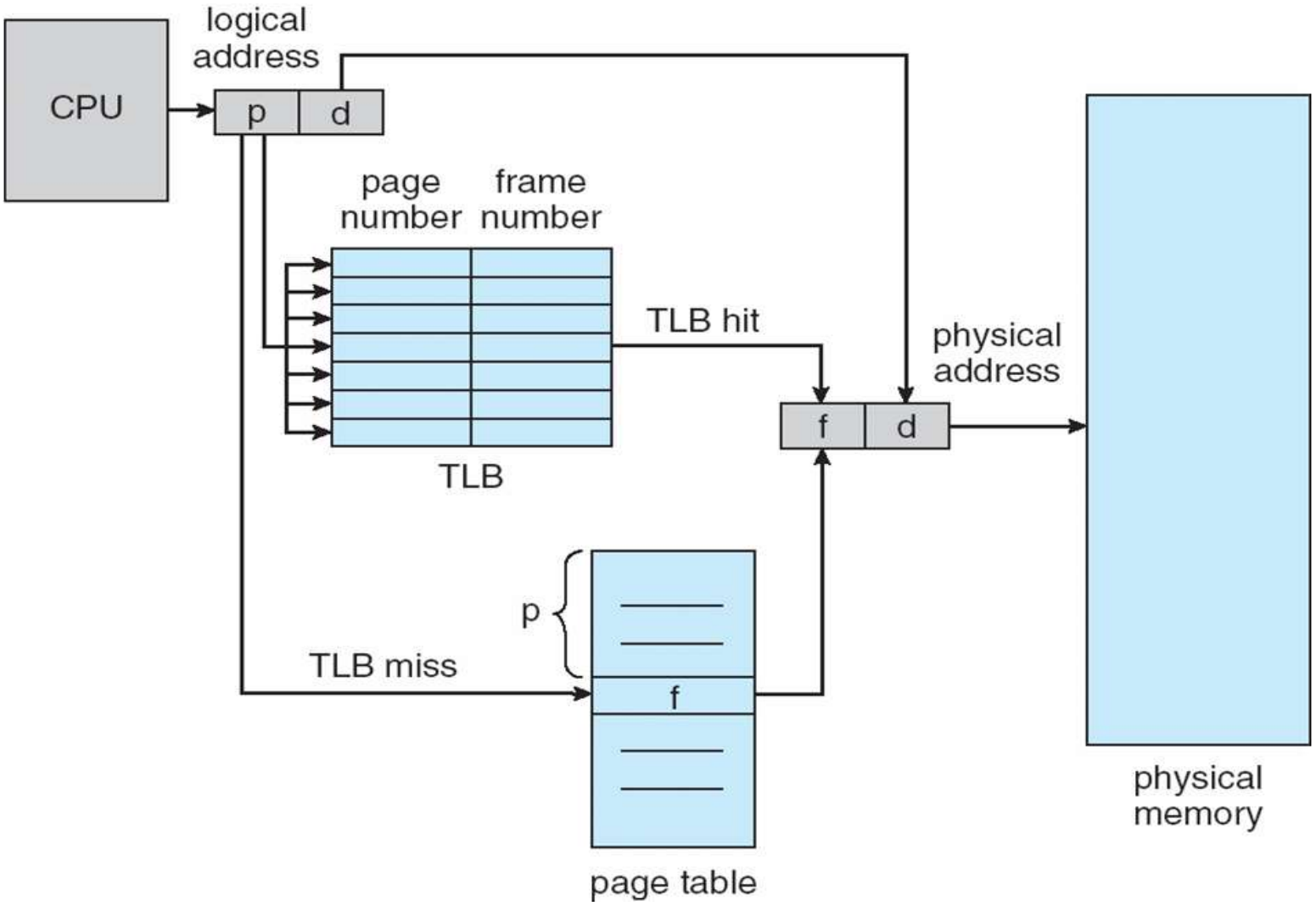
# Associative Memory

- 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

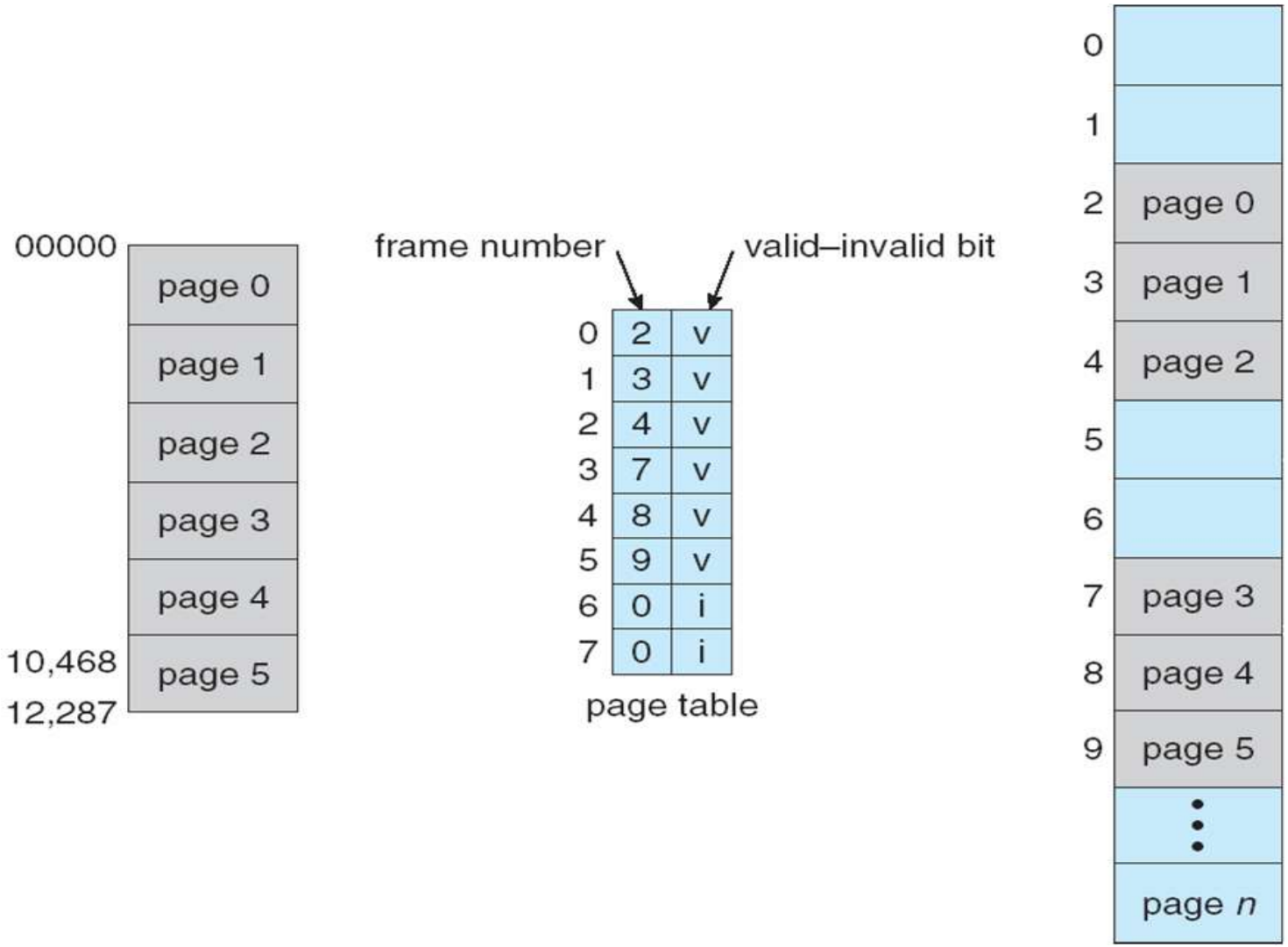
# Paging Hardware With TLB



# 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

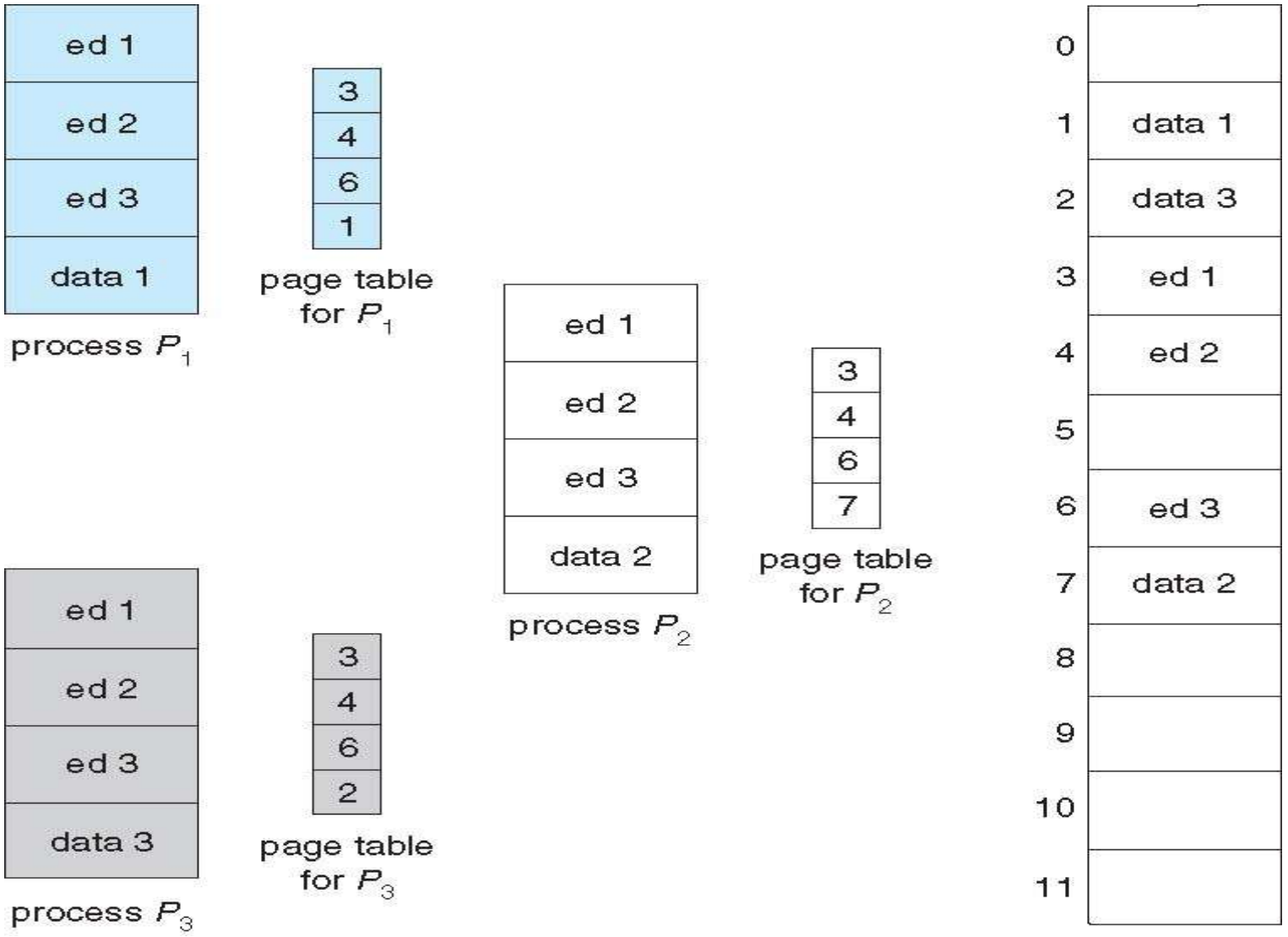
# Valid (v) or Invalid (i) Bit In A Page Table



# Shared Pages

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



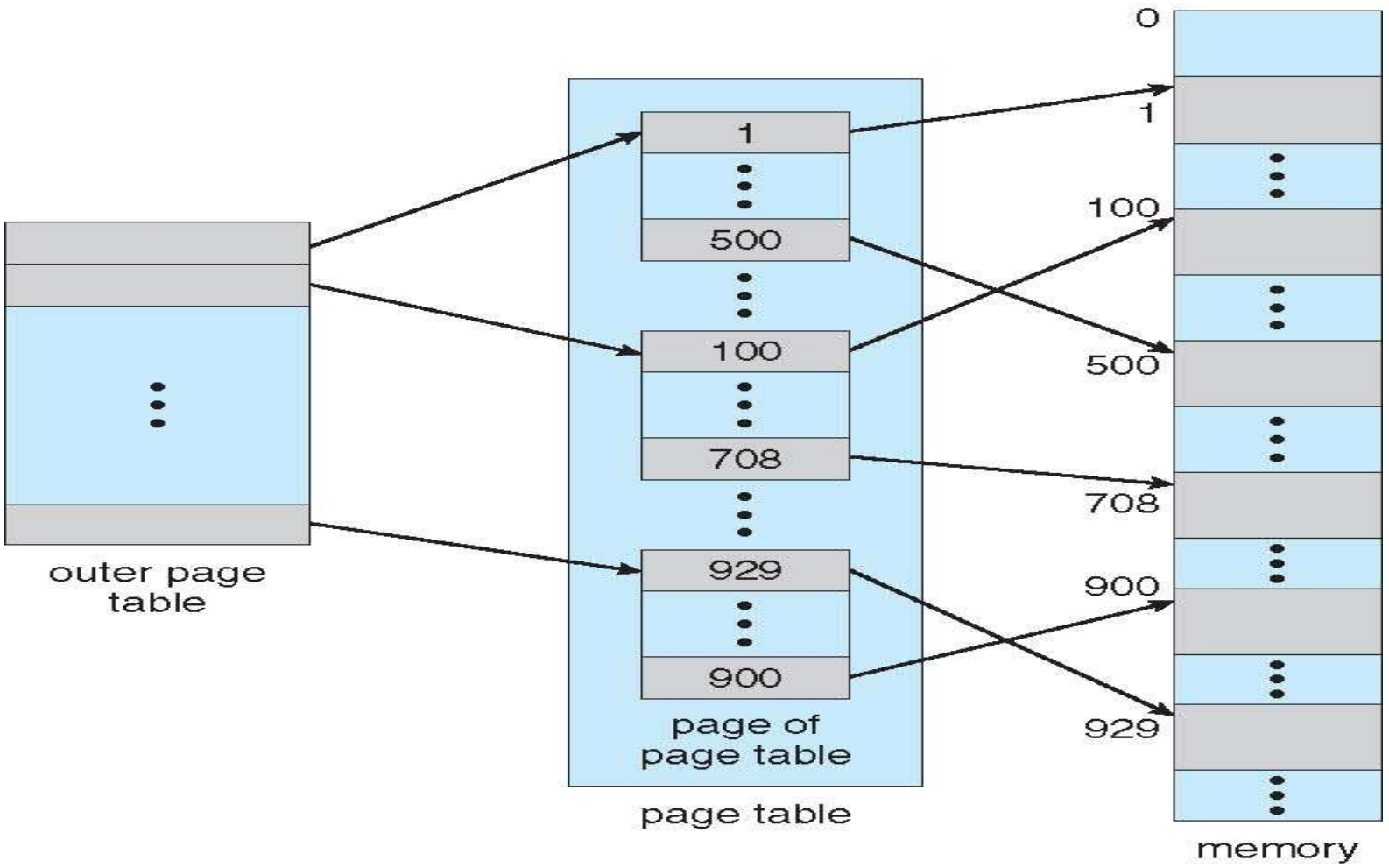
# Structure of the Page Table

- 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 ($2^{32}$ / $2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Page Tables
- Hashed Page Tables
- Inverted Page Tables

# 1. 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

# Two-Level Page-Table Scheme

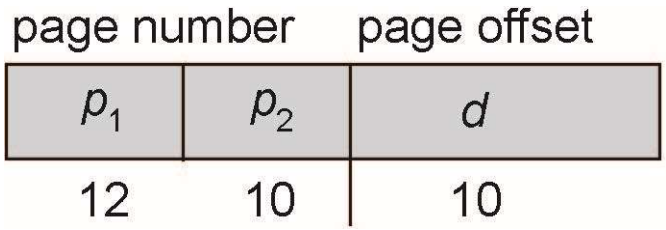


# Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits

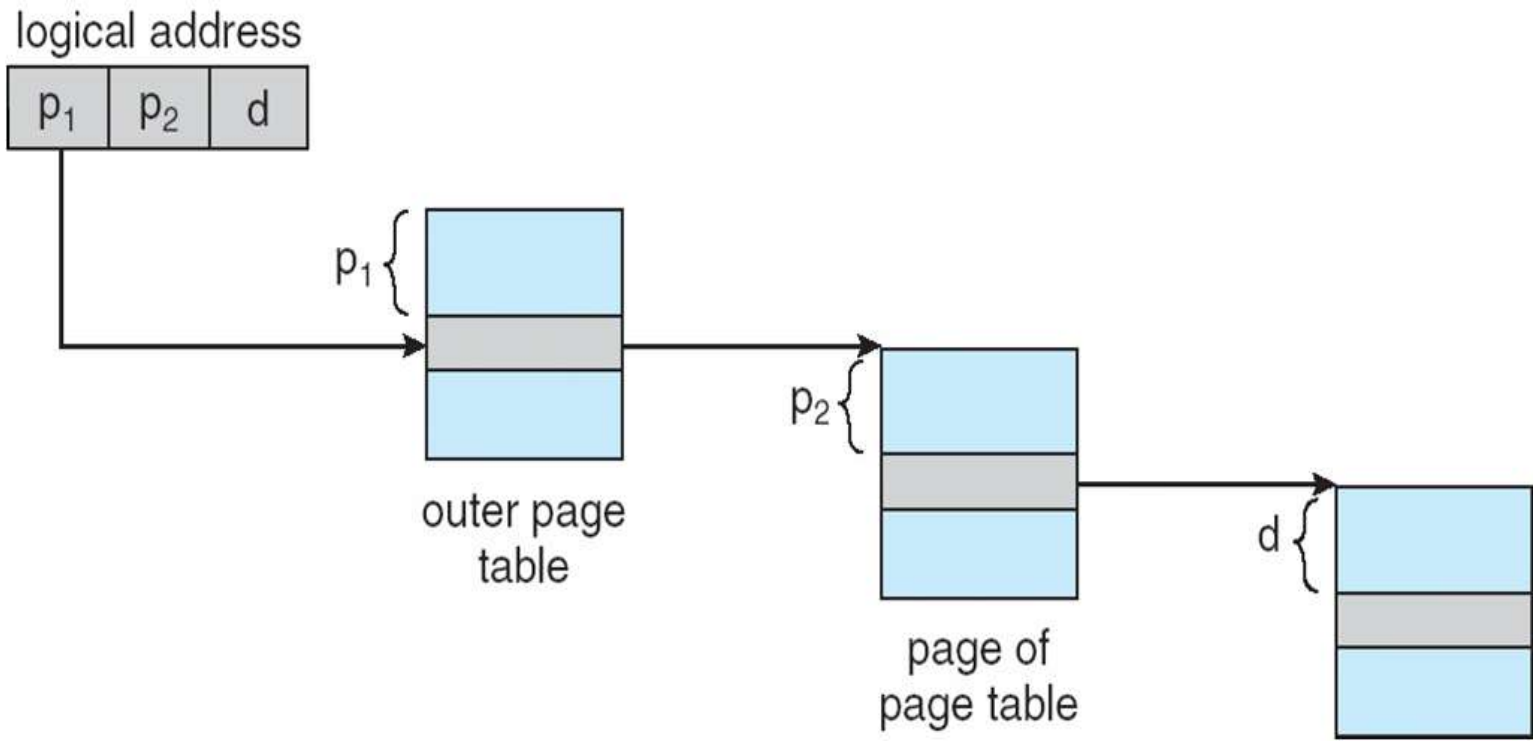
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset

- Thus, a logical address is as follows:



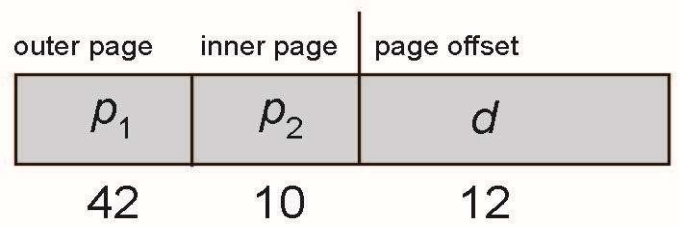

- where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

# Address-Translation Scheme



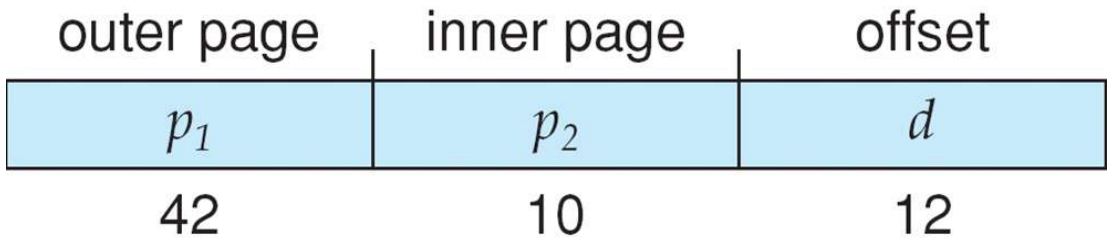
# 64-bit Logical Address Space

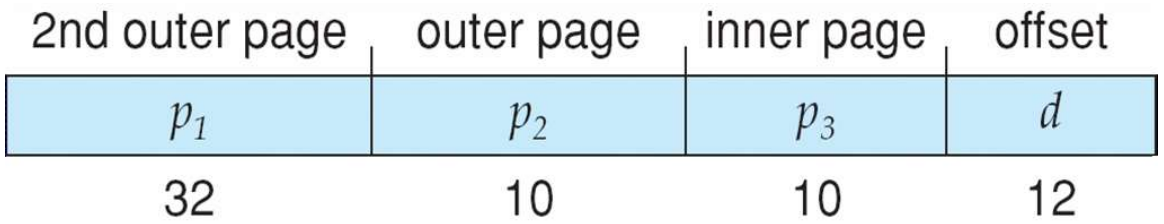
- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
  - Then page table has $2^{52}$ entries
  - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  - Address would look like




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

# Three-level Paging Scheme

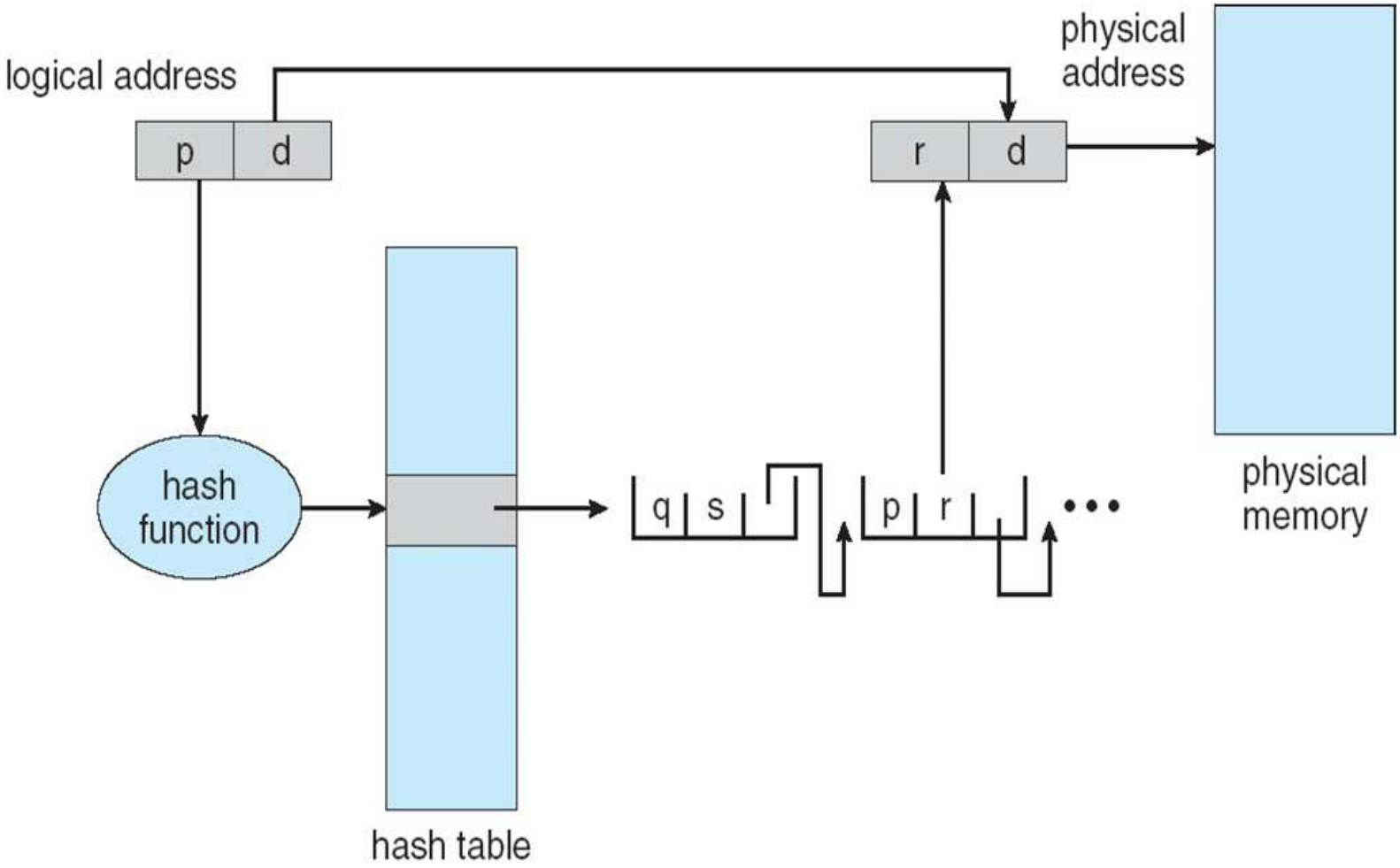


# 2. Hashed Page Table

- 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)

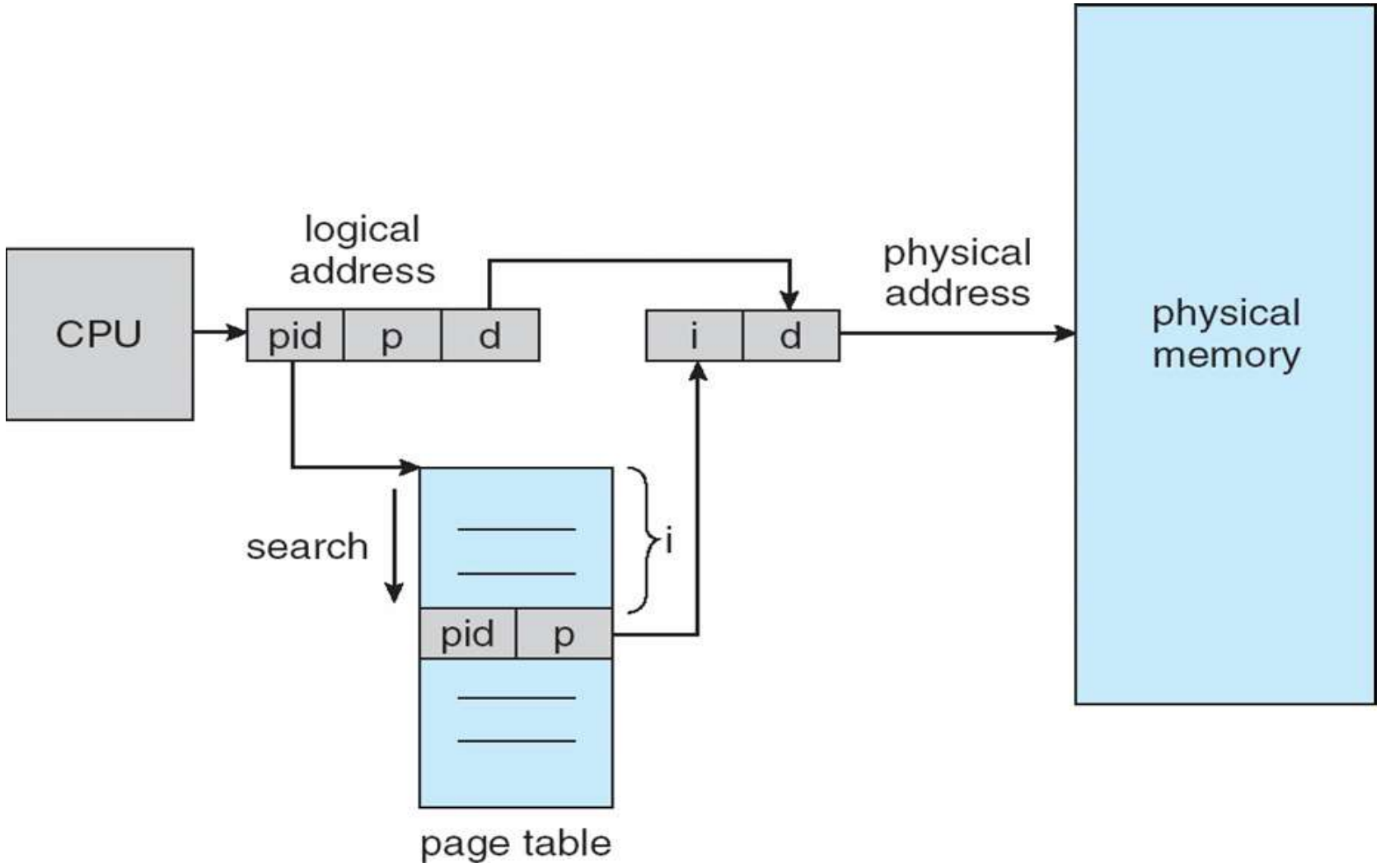
# Hashed Page Table



# 3. 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

# Inverted Page Table Architecture



# Numericals

## Memory Management

# Page Table Size

- The size of a page table is given by the following equation:
- $$Page\ table\ size = \frac{logical\ address\ space\ size \times page\ table\ entry\ size}{page\ size}$$
- For example:
  - Many processors have 32-bit logical addresses, which results in a 4GB logical address space size. The page table entries size is usually 4B. If the page size is 4KB then the page table size is:
  - $$Page\ table\ size = \frac{4\ GB \times 4\ B}{4\ KB} = 4MB$$

# Paging – Effective Access Time

Find the effective access time when there is an 80% chance of a TLB hit and it takes 100 nanosecond to access the memory.

EAT = Effective Access Time =

(tlb_hit_ratio x memory access time) +
(tlb_miss_ratio x 2*memory_access_time)

= (0.80 x 100) + (0.20 x 200) = 120 nanoseconds

# Memory Management – Memory size

- Suppose you have a ram of size 32MB, calculate the total number of locations/entries in ram and the number of bits required to address each location.

**Solution:** Supposing the system is byte addressable.

Memory size = 32MB = loc x 1B (where loc is the number of locations available in memory supposing each location is of 1 byte)

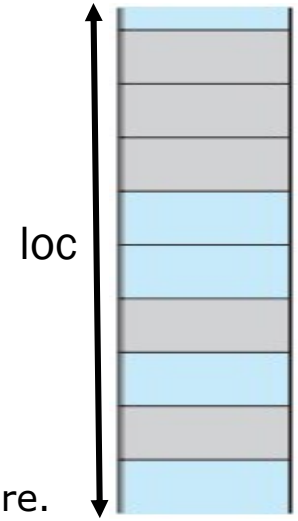
loc = 32MB/1B = 32M locations = $32 \times 2^{20} = 2^5 \times 2^{20}$

(we divide memory size by the size of each location to get # of locations)

$= 2^{25}$

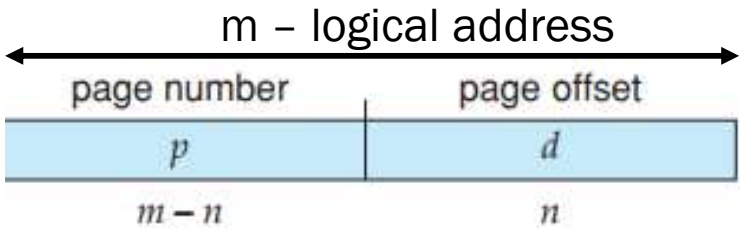
So the total number of bits required to represent $2^{25}$ locations is = 25

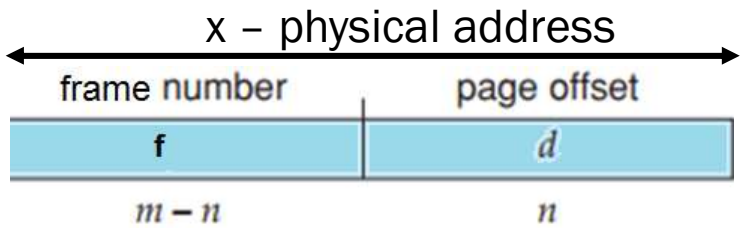
Note: You maybe provided with the size of each location i.e. 1B assumed here. Systems can be byte or word addressable in general.



# Memory Management - Paging

- Supposing logical address = 26 bits
- Physical address = 16 bits
- Page size = 1KB
- **Find the following: # of frames & pages**




Supposing system is Byte addressable. Now consider the following calculations:

Since the page size is 1KB. So, 1KB = $2^{10}$ B

So the offset n = 10 bits.

Since x is the physical address space, and n is number of offset bits. x – n = physical address location bits. 16 bits – 10 bits = 6 bits. Hence, $2^6$ is the total number of frames.

m is the logical address space. As n = 10, m – n = 26 – 10 = 16. Hence $2^{16}$ = 64K pages in the virtual address space

# Memory Management - paging

- Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.
  - a. How many bits are required to represent the logical address space?
  - b. How many bits are there in the physical address?

Logical address space bits = m = ?

Recall that the size of the page/frame is = $2^n$ and the size of logical address space are $2^m$

Then, **logical address space = # of pages x page size**

➔ $2^m$ = # of pages x $2^n$

➔ # of pages = 64 = $2^{m-n}$

n = ?

Page size = $2^n$ ➔ 1024 = $2^n$

$2^{10}$ = $2^n$ ➔ n = 10-bits

Again: # of pages = $2^{m-n}$ ➔ 64 = $2^{m-10}$ ➔ $2^6 = 2^{m-10}$ ➔ 6 = m -10 ➔ m = 16-bits (logical address)

Physical address: Let x be the physical address bits.

Size of physical address space = $2^x$

Size of physical address space = # of frames × frame size (frame size = page size )

Size of physical address space = 32 × 1024 ➔ $2^x = 2^5 \times 2^{10}$ ➔ $2^x = 2^{15}$ ➔ number of bits = x =15 bits

# Memory Management - paging

- Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.
  - a. How many bits are required to represent the logical address space?
  - b. How many bits are there in the physical address?

**Easy solution**

Addressing within a 1024-words page requires 10 bits because $1024 = 2^{10}$. Since the logical address space consists of 64 pages = $2^6$ pages, the logical addresses must be 10+6 = 16 bits. Similarly, since there are 32 frame = $2^5$ physical frames, physical addresses are 5 + 10 = 15 bits long.

# Memory Management - paging

Consider a system using Paging for memory management. The system uses a page size of 1024 bytes, and the length of the address register is 18 bits. Now compute the following parameters:

a) Size of RAM
b) Total number of frames in the RAM
c) Number of bits in a page table entry

**Solution:**

a) $2^{18} = 2^{8} * 2^{10} = 256 * 1024 = 256$ KB

b) 256 KB / 1024 B = 256

c) 18 - 10 = 8 bits

# Memory Management - Paging

**Question**

Write a C/C++ function to perform address translation in a system using Paging for memory management. You are given with the page size, page table, and a logical block no, and you are required to compute the corresponding physical address. Use the following named constant, and the function prototype:

```
const int PS = ... ;          // page size

int translate(int log,        // logical block no
              int tbl[],      // page table
              int n);         // length of the page table
```

**Solution:**

```
int translate(int log, int tbl[], int n) {

     int page = log / PS;        // compute page no
     int frame = tbl[page];      // get frame no
     int off = log % PS;         // calculate offset within the page

     return frame * PS + off;    // compute the physical address
}
```

# Memory Management - Paging

Consider a system with a page size of 256. Assume a process running in this system has the following page table:

| Page number | Frame number |
|---|---|
| 0 | 50 |
| 1 | 10 |
| 2 | 90 |

Now translate the following logical addresses into the corresponding paging physical addresses:

i) 700

page = 700 / 256 = 2

frame = tbl[2] = 90

offset = 700 mod 256 = 188

phy add = 90 * 256 + 188

= 23,228

# Memory translation – logical to physical

Given a logical address of 2486 and a page size of 256. What is the corresponding physical address?

We can divide the address by page size to get page #. So 2486/256 = 9.7 Therefore, the page # is 9. In the page table, we can grab the corresponding frame number.

In order to find the offset, we take the mod of 2486 with 256. So, 2486%256 = 182

How to find the exact address?
Supposing that the page table returned 5 as the frame number against the page number 9. The exact memory address of the frame will be:
(locations/entries_Per_Page*frame_Number)-1 (-1 because memory indexing starts at 0)

So ➔ (256 * 5) – 1 = 1279
Now, we will add the offset to it to get the actual physical memory address:
1279 + 182 = 1461 = 0x5B5 is the memory location.

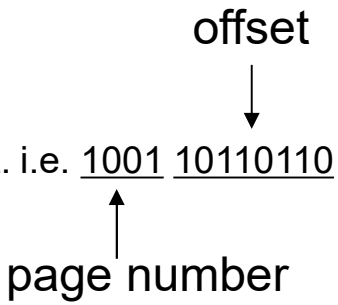
# Memory translation – logical to physical(2nd method)

**Given a logical address of 2486 and a page size of 256. What is the corresponding physical address?**

Convert the logical address to binary number. 2486 in binary is: 100110110110
Now, take the $\log_2$ of the page size. i.e. $\log_2(256) = \log_2(2^8)$ ➔ 8

In the above binary number, the least-significant 8 numbers are the page offset. i.e. 1001 10110110
The underlined 0's and 1's is the offset. 10110110 = 182

The most significant left most 4 bits represents the page number.




Page number is therefore 9.

How to find the exact address?
Supposing that the page table returned 5 as the frame number against the page number 9. The exact memory address of the frame will be: (locations/entries_Per_Page*frame_Number)-1 (-1 because memory indexing starts at 0)

So ➔ (256 * 5) – 1 = 1279
Now, we will add the offset to it to get the actual physical memory address: 1279 + 182 = 1461 = 0x5B5
Is the memory location.

# Paging guidance

***Paging arithmetic laws:***

Page size = frame size

Logical address space (/size) = $2^m$

Logical address space (/size) = # of pages × page size

Physical address space (/size) = $2^x$ (*where **x** is the number of bits in physical address*)

Physical address space (/size) = # of frames × frame size

Page size= frame size= $2^n$

# of pages= $2^{m-n}$

# of entries (records)in page table = # of pages

# References

- Chapter 3, Modern Operating System
- http://cseweb.ucsd.edu/classes/sp00/cse120_A/mem.html
- Operating System Concepts (Silberschatz, 9th edition) Chapter 8