Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Main Memory-Part2

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2022

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**

Paging (cont.)

- Keep track of **all free frames**

To run a program of size N pages

↓

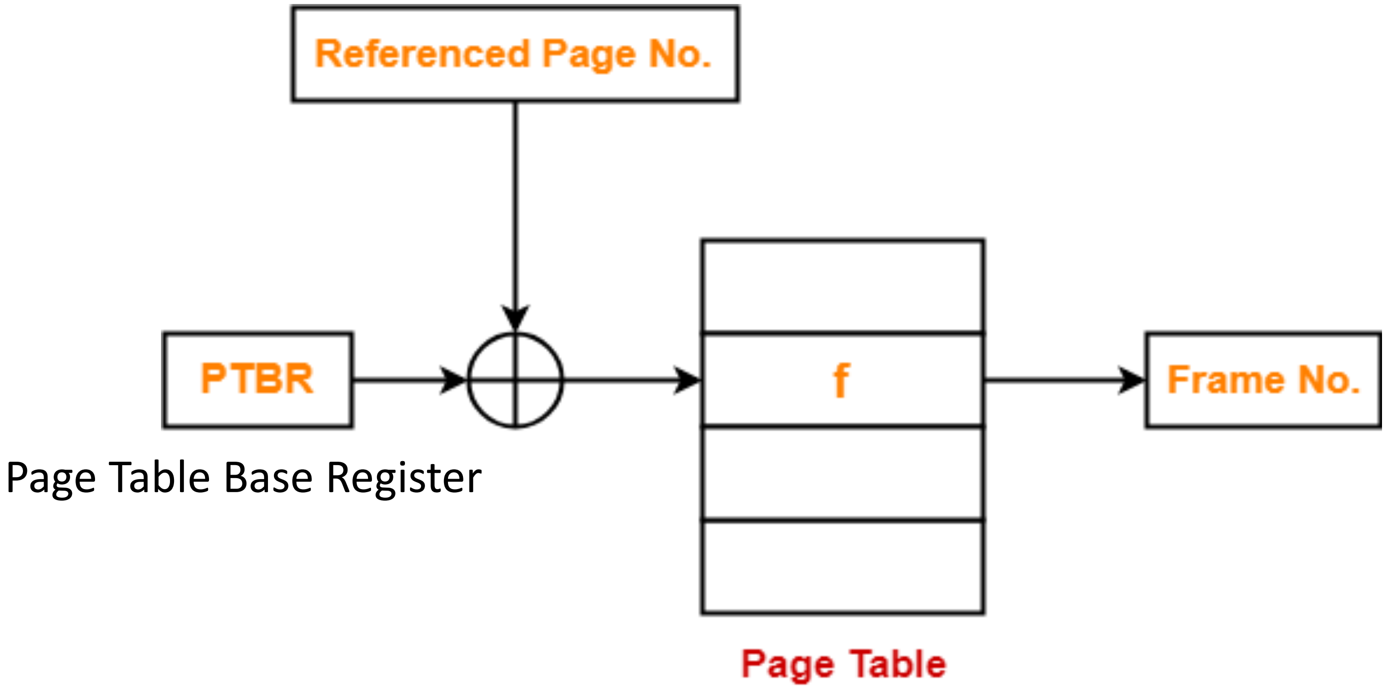
find N free ***frames***

↓

Load program

Paging (cont.)

- Set up a **page table** to translate logical to physical addresses

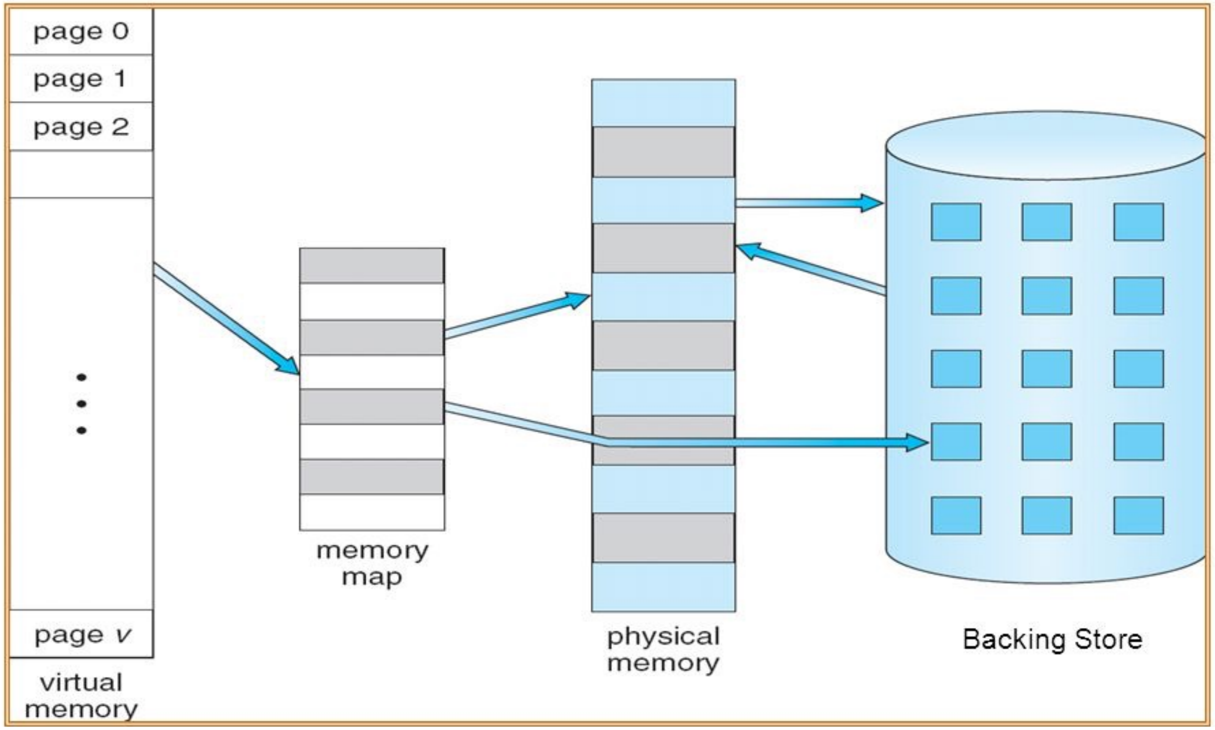


Obtaining Frame Number Using Page Table

Source: https://www.gatevidyalay.com/page-table-paging-in-operating-system/

Paging (cont.)

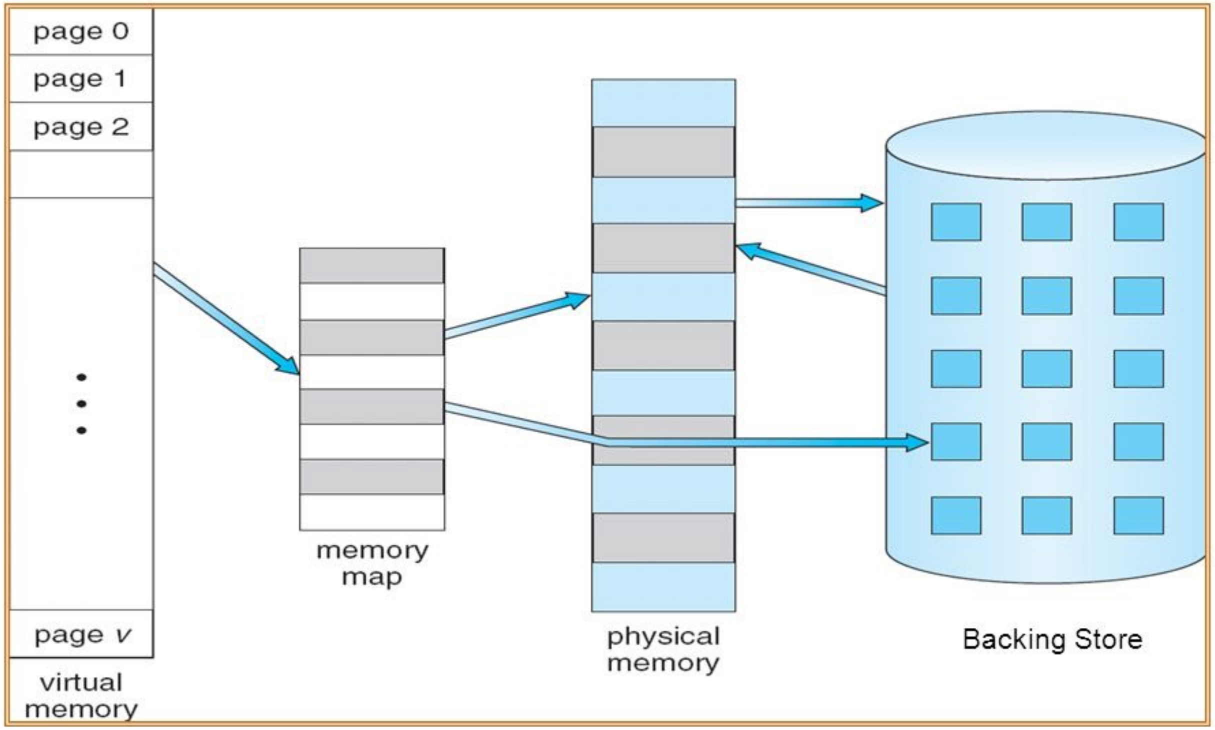
- **Backing store** likewise split **into pages**



- **Does paging solve fragmentation issue?**

Paging (cont.)

- **Backing store** likewise split **into pages**



- **Still have Internal fragmentation**

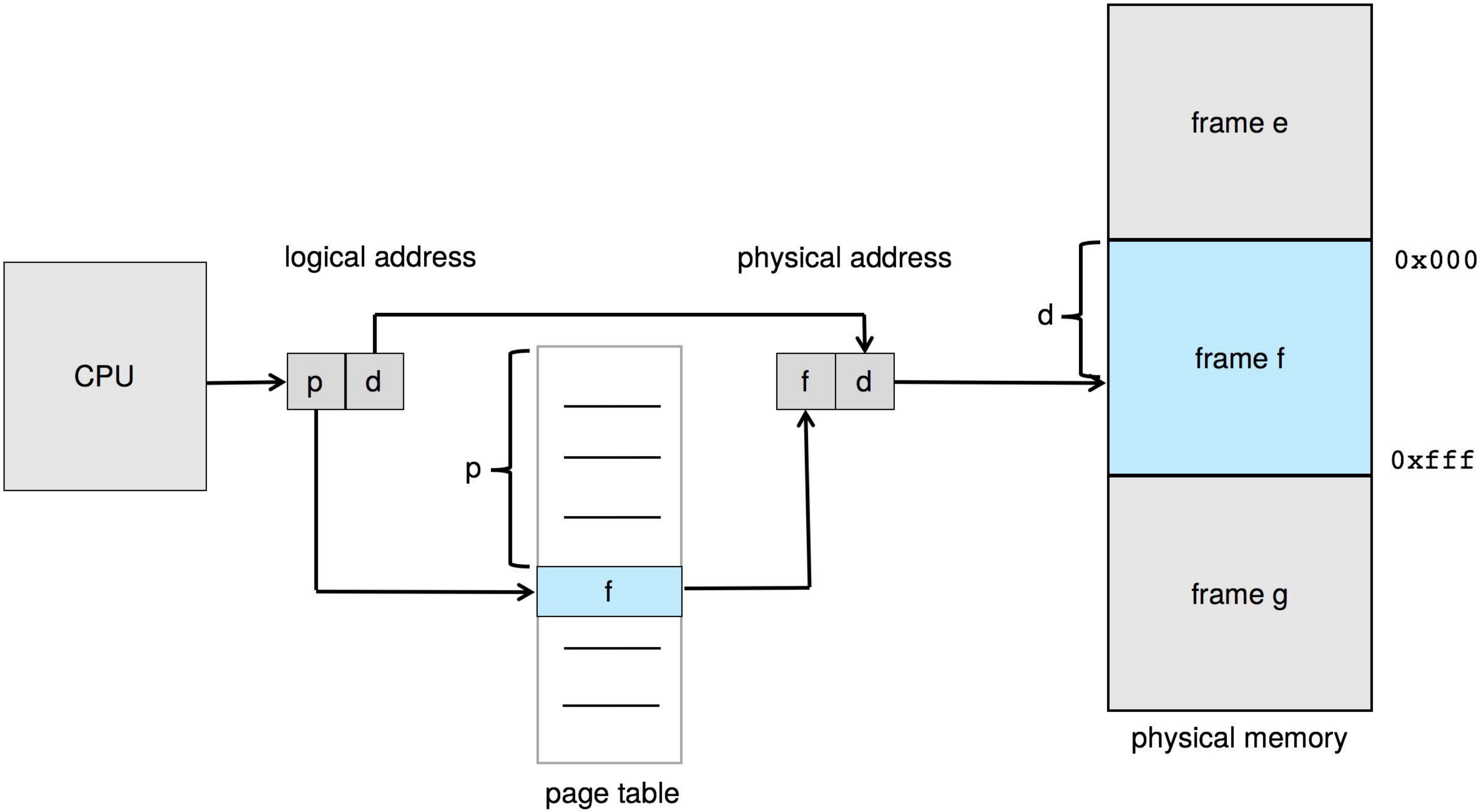
Address Translation Scheme

- 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

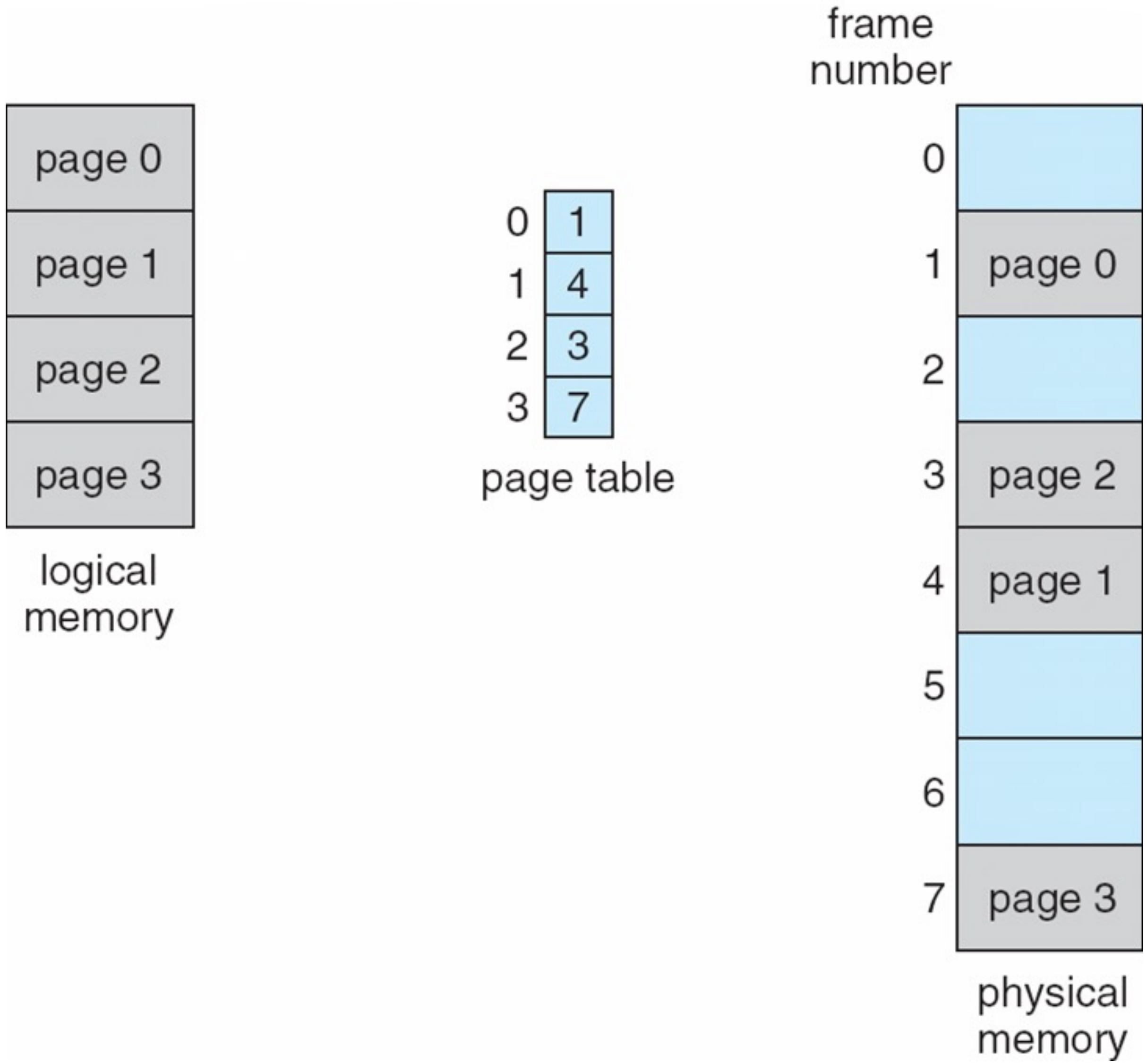
page number	page offset
p	d
m -n	n

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

Paging Hardware

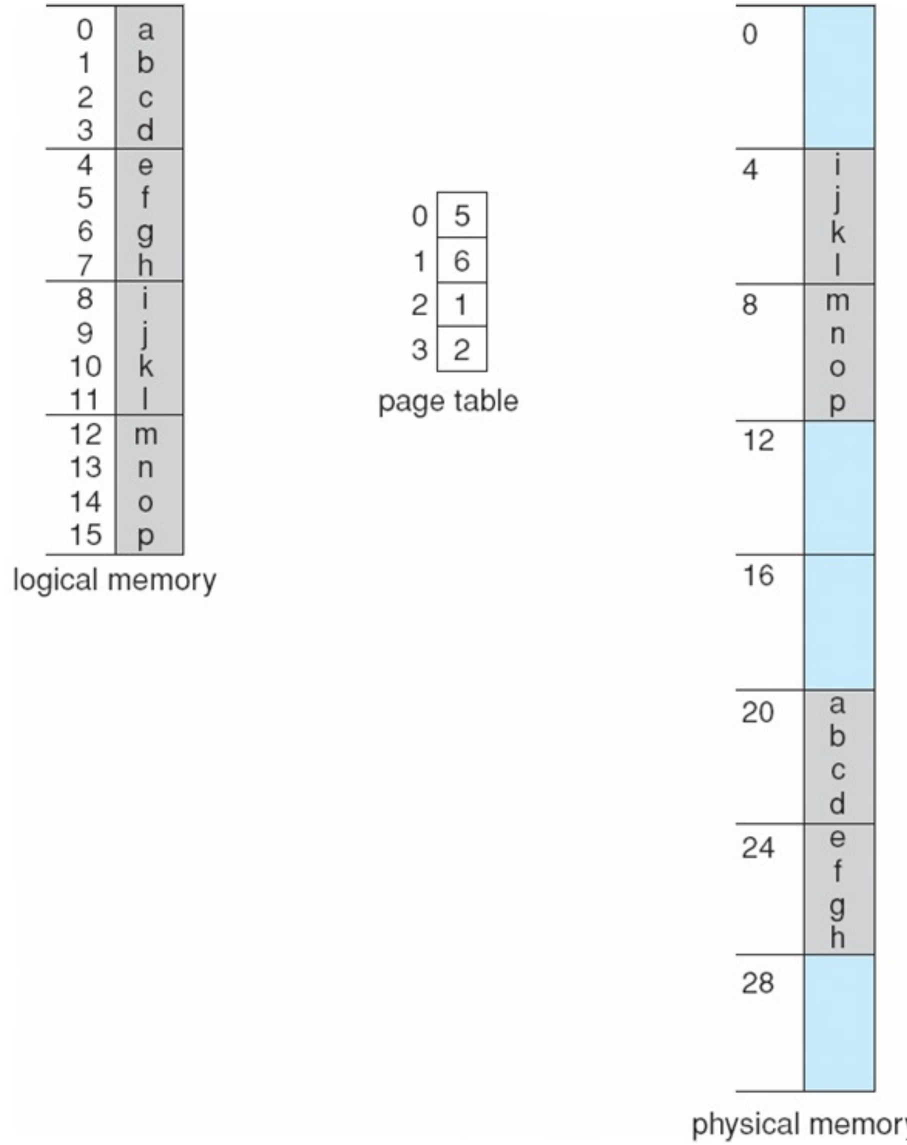


Paging Model of Logical and Physical Memory



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)



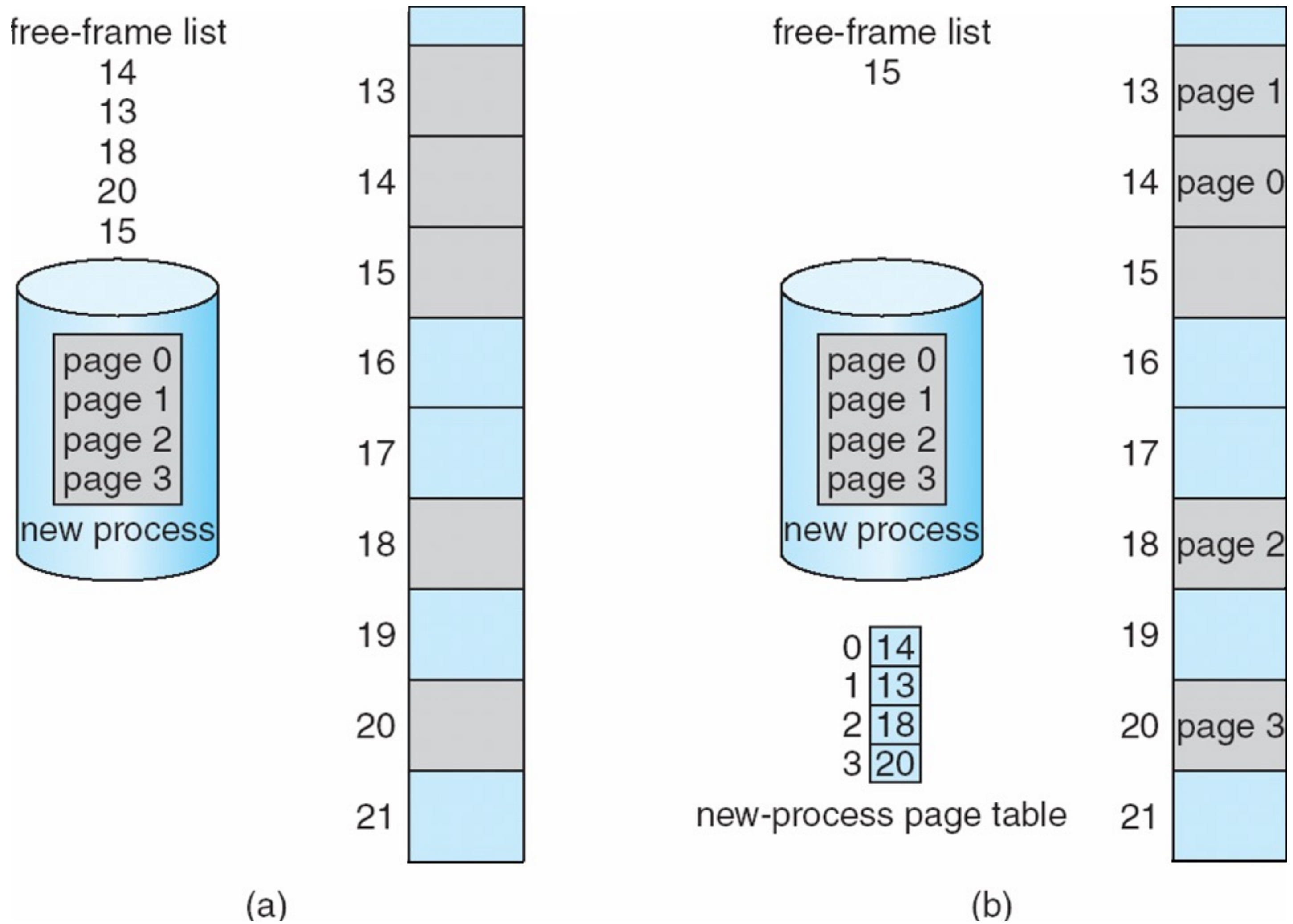
Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of ?
- Worst case fragmentation = ?
- On average fragmentation = ?

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

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
  - One for the data / instruction

Implementation of Page Table

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

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.
  - What if ASIDs is not supported?
    - Otherwise need to flush at every context switch.

Translation Look-Aside Buffer

- 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

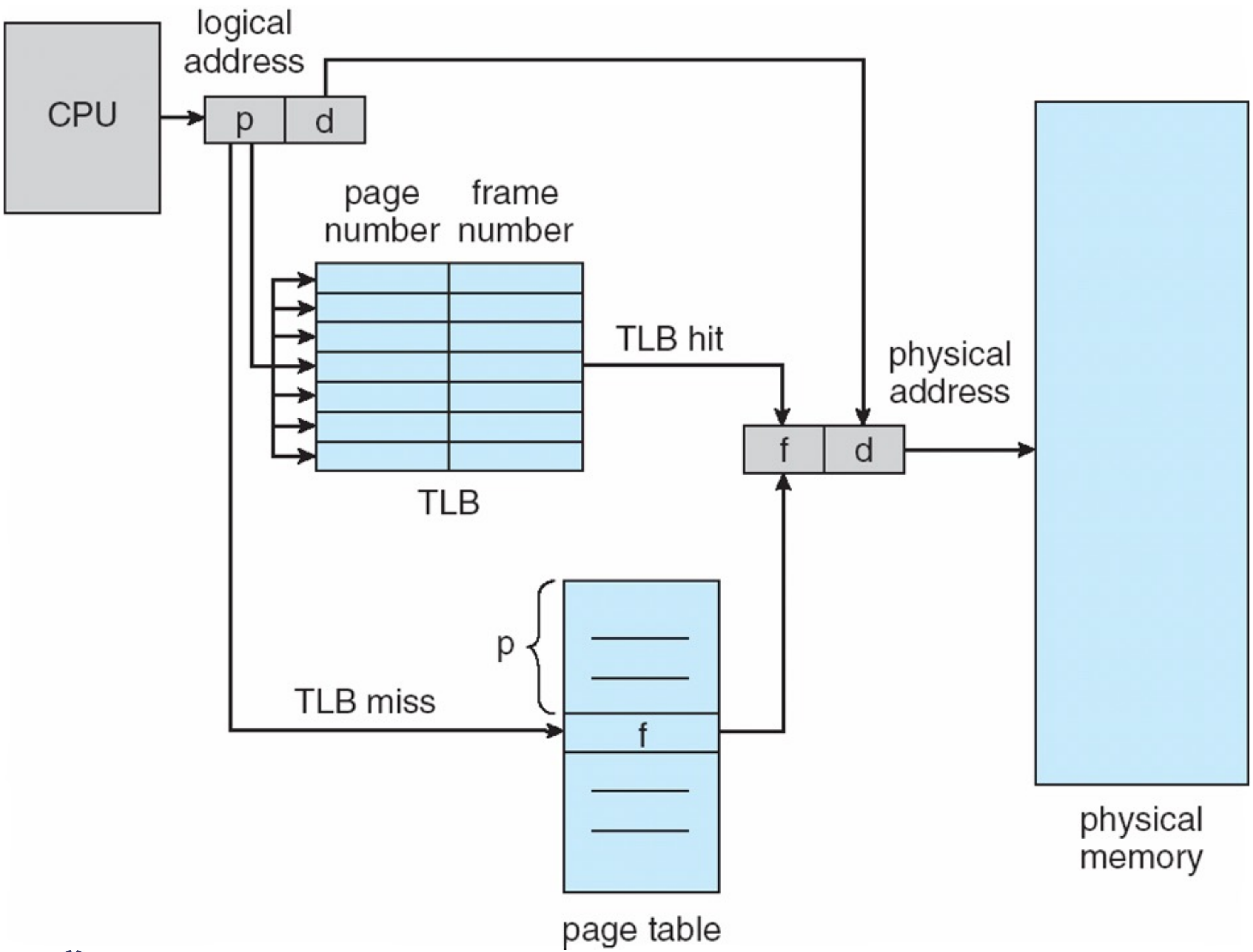
Hardware

- 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

Paging Hardware With TLB



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.

- Suppose that 10 nanoseconds to access memory.
  - If the desired page in TLB then a mapped-memory access take 10ns.
  - Otherwise, we need two memory access so it is 20 ns

Effective Access Time (cont.)

- **Effective Access Time (EAT)**

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

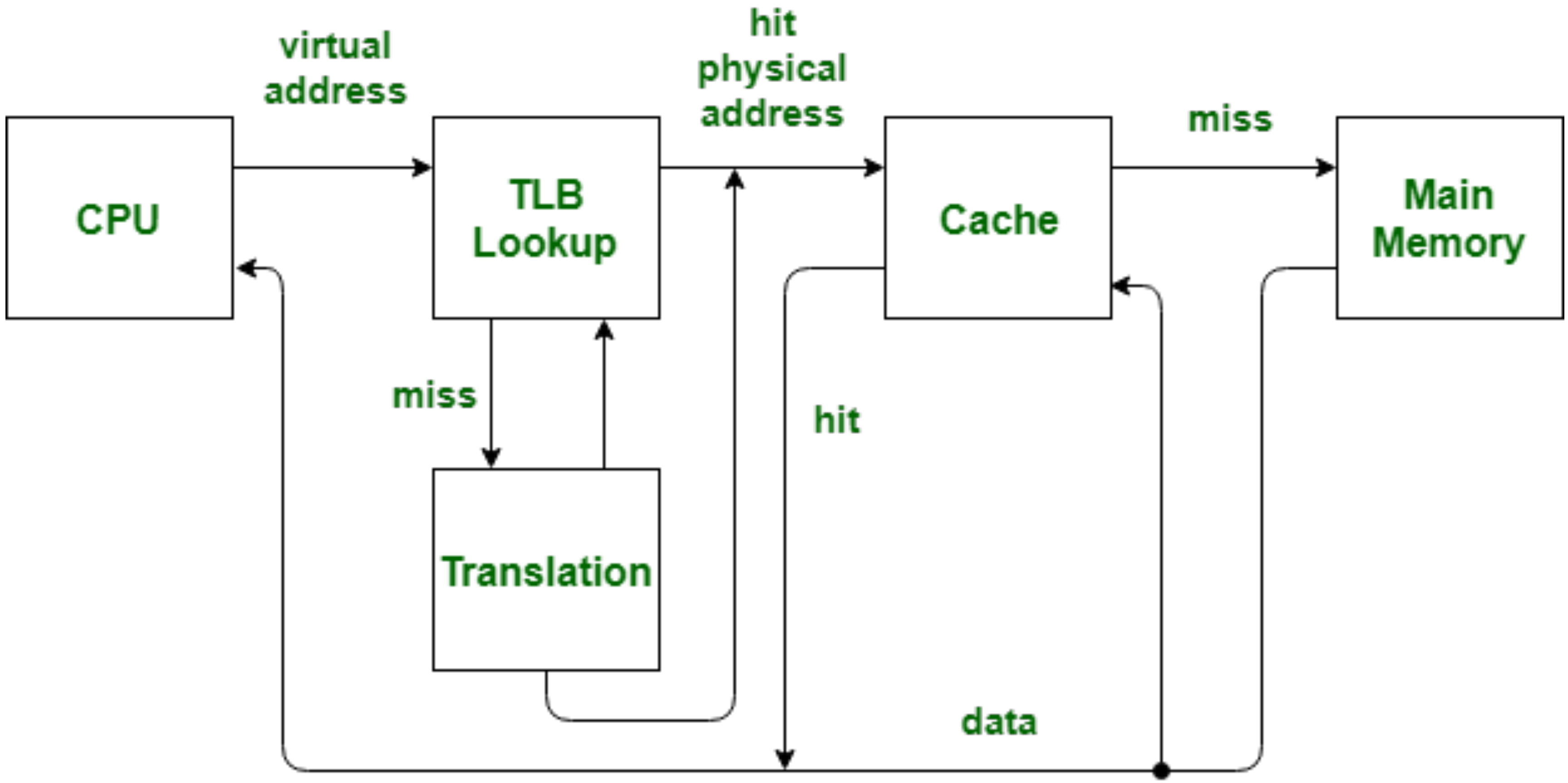
  implying 20% slowdown in access time

- Consider a more realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

Cache vs TLB



- https://stackoverflow.com/questions/1973473/difference-between-cache-and-translation-lookaside-buffertlb
- https://www.geeksforgeeks.org/whats-difference-between-cpu-cache-and-tlb/

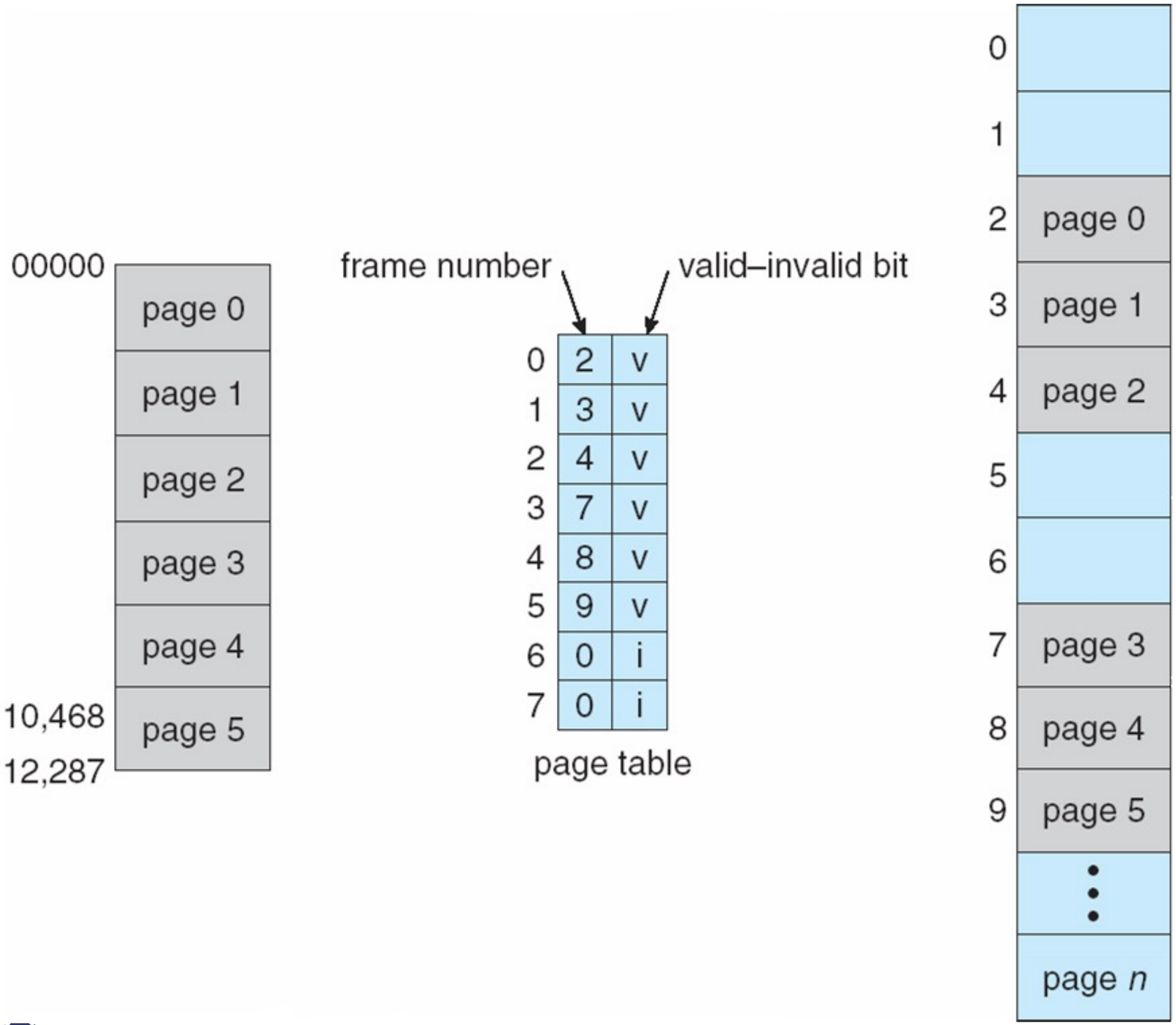
Memory Protection

- 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

Memory Protection

- Valid-invalid bit attached to each entry in the page table:
  - **valid**: the associated page is in the process' logical address space, and is thus a legal page
  - **invalid**: 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

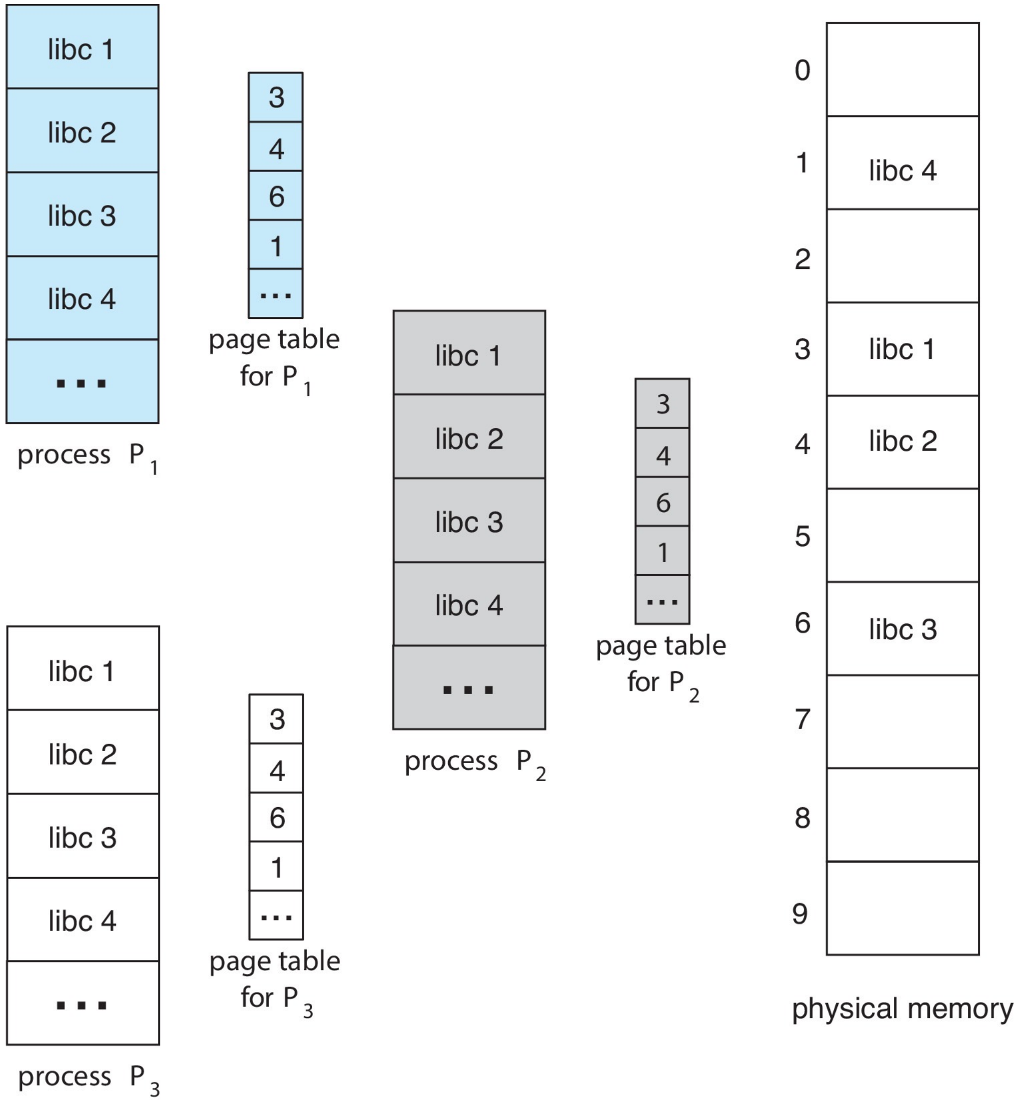
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

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

Shared Pages Example



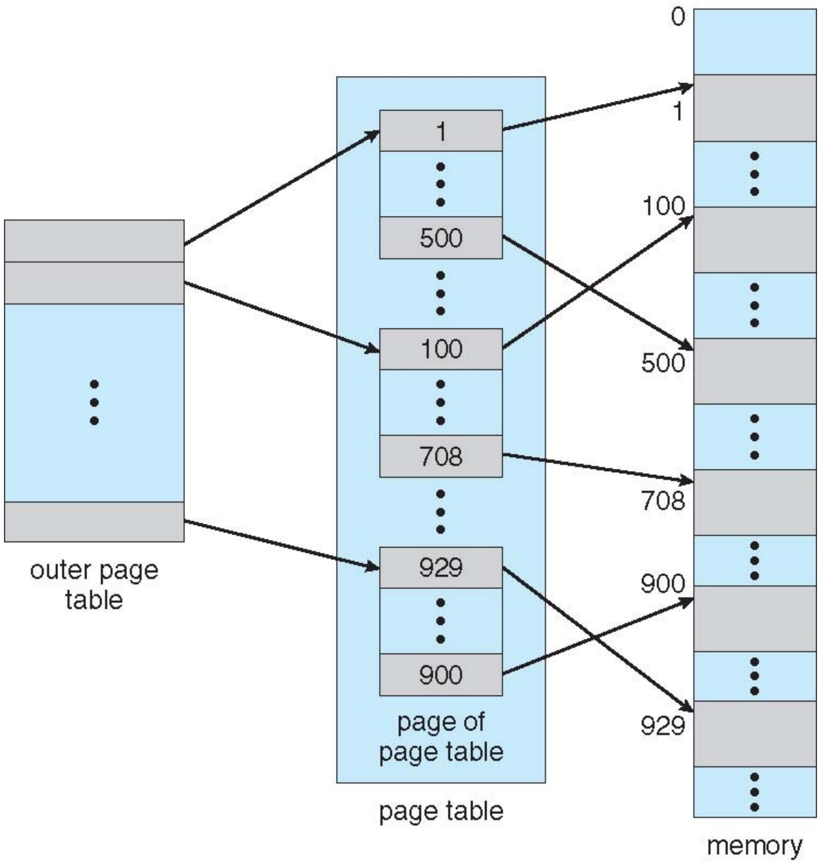
Shared Pages

- **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.

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



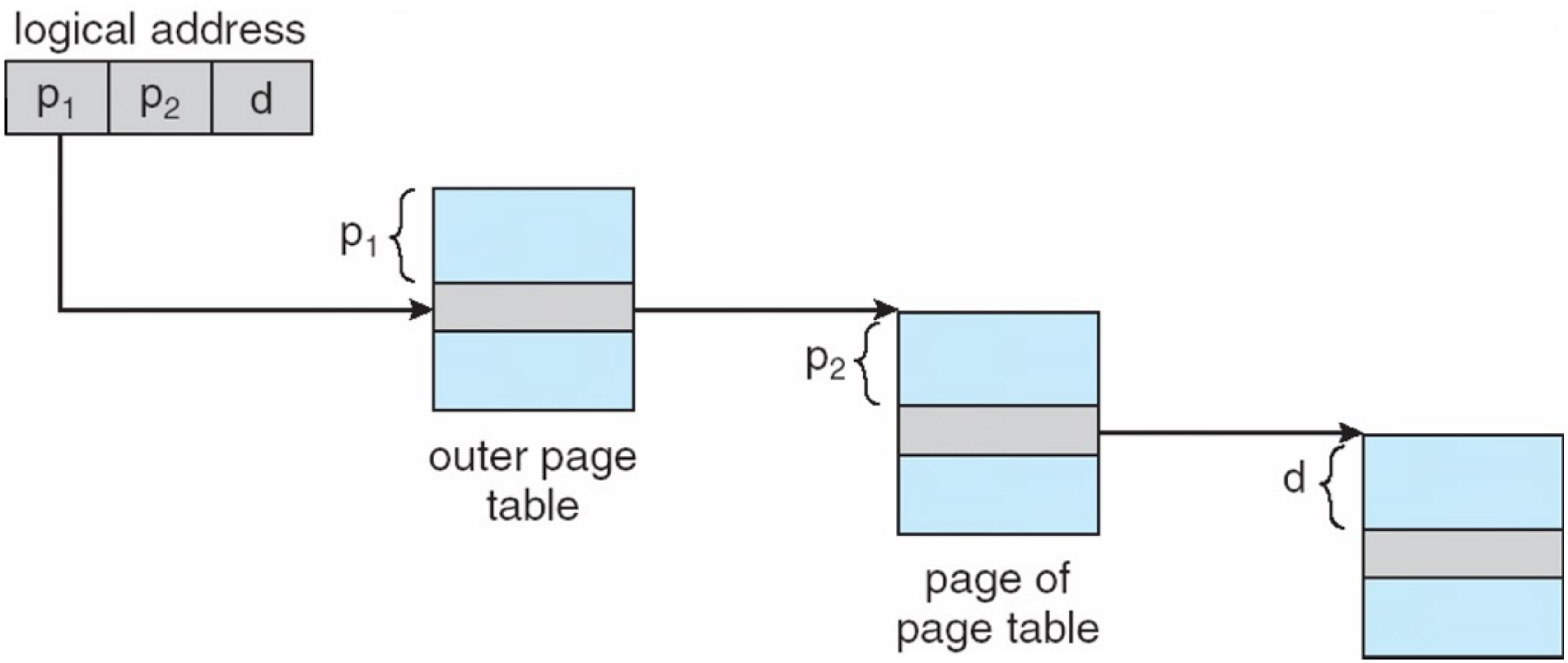
Two-Level Paging Example

- 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**

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	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

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

In general, is it ***appropriate*** to use hierarchical page tables for 64 bit architecture?

x86-64 example

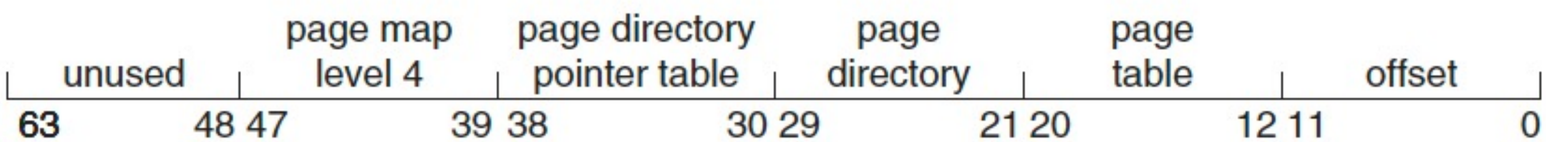


Figure 9.25 x86-64 linear address.