CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 29

Main Memory

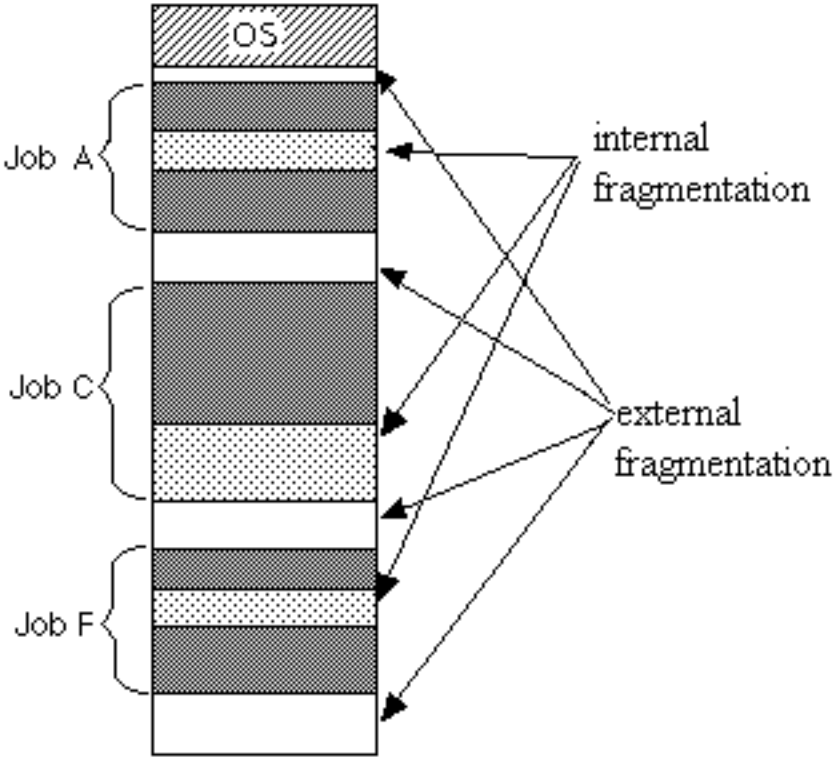
Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

FAQ

Terminology:

- Contiguous allocation: Allocation of a contiguous memory region to a process (from address x to x+y)
- Internal vs external fragmentation
  - Internal fragmentation: memory wasted within an allocated memory region
  - External fragmentation: memory wasted due to small chunks of free memory interspersed among allocated regions



FAQ

Paging:

Can we use really small pages/frames to minimize internal fragmentation within frames?

- Answer coming up

Paging and Addresses

- Part of the address identifies a page
- The other part identified location within the frame
- More coming up

Pages

- Pages and frames
- Page tables
- TLB: page table caching
- Memory protection and sharing
- Multilevel page tables

Paging

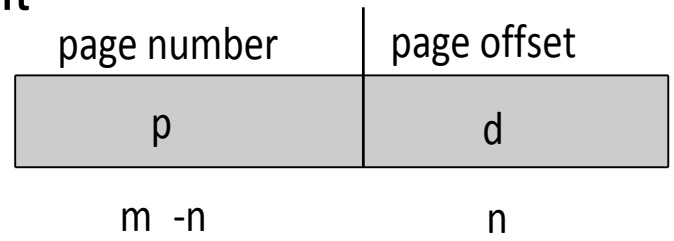
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Still have Internal fragmentation
- 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

- physical memory - **frames**
- logical memory - **pages**
- Keep track of all free frames
- Set up a **page table** to translate logical to physical addresses

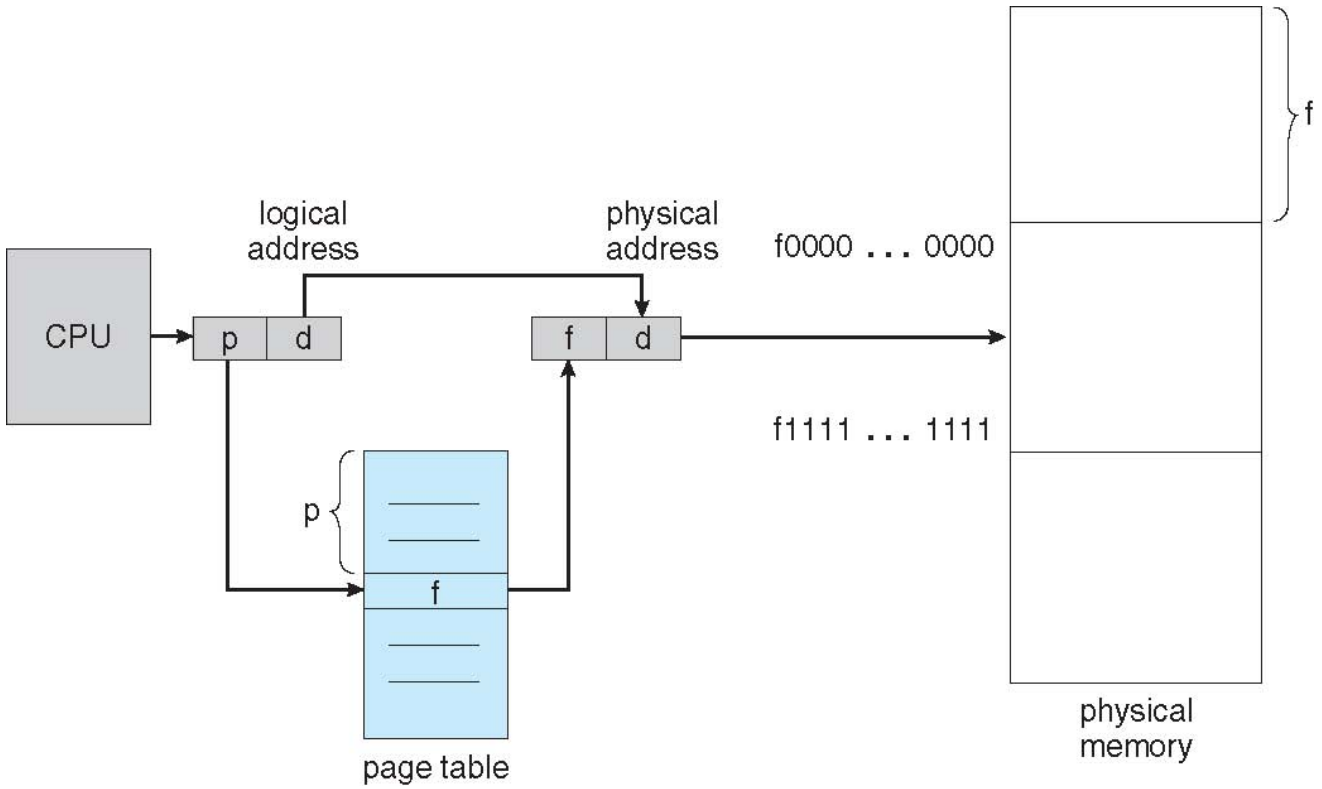
Address Translation Scheme

- 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



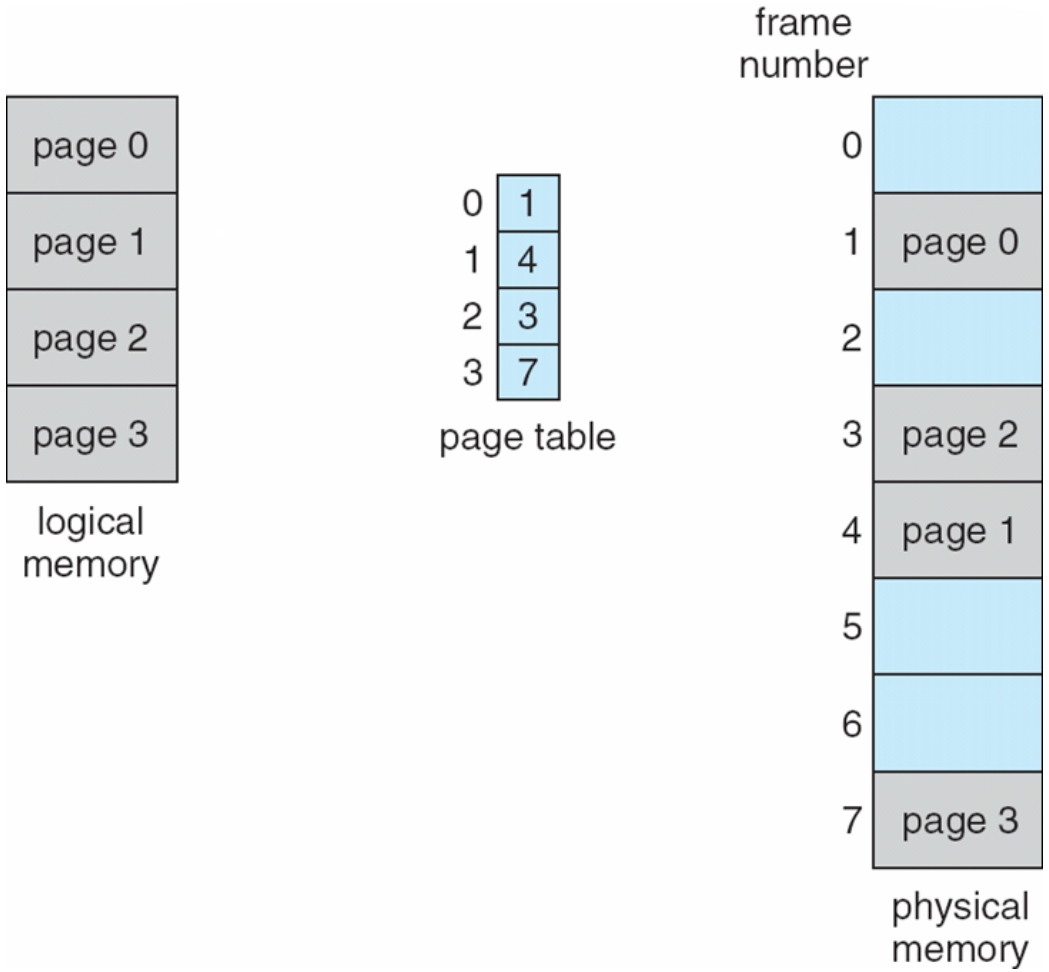
  - For given logical address space 2^m and page size 2^n

Paging Hardware

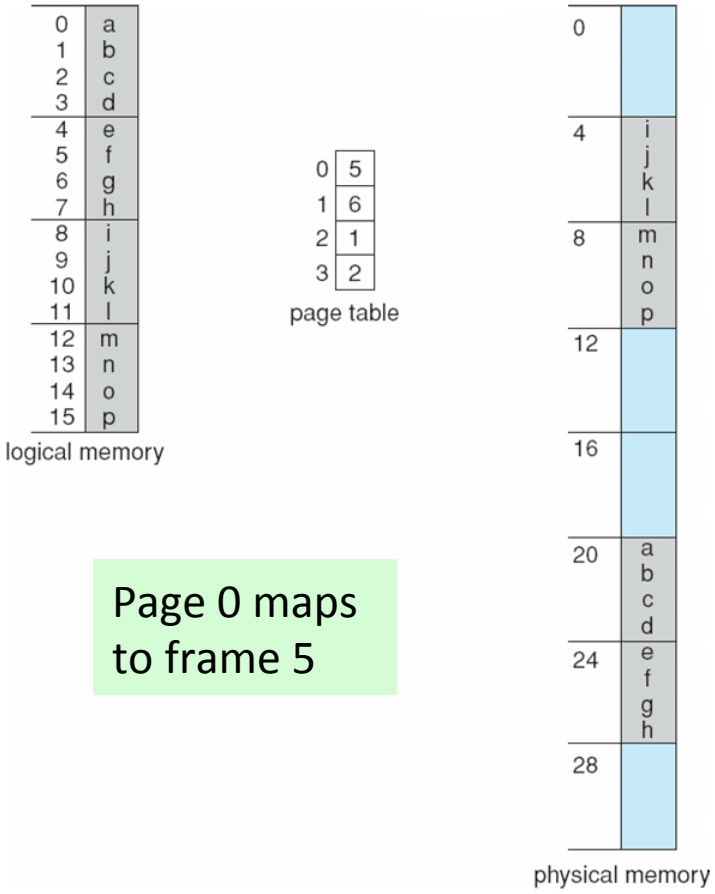


Page number p to frame number f translation

Paging Model of Logical and Physical Memory



Paging Example



8 frames
Frame number 0-to-7

Page 0 maps to frame 5

n=2 and m=4 Logical add. space = 2^4 bytes, 2^2=4-byte pages

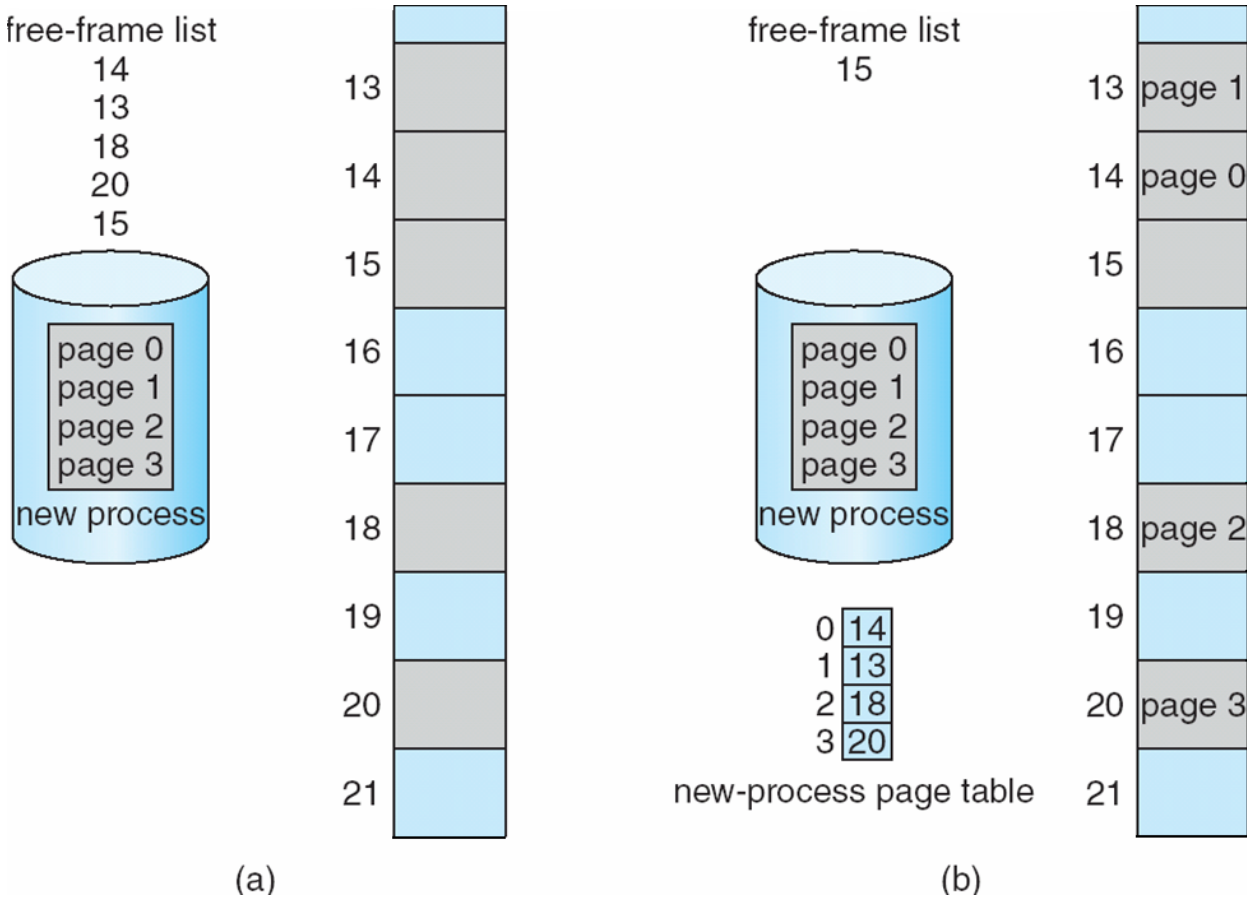
32-byte memory with 8 frames

Colorado State University

Paging (Cont.)

- Internal fragmentation
  - Ex: 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 size growing over time
    - X86-64: 4 KB (common), 2 MB ("huge" for servers), 1GB ("large")
- Process view and physical memory now very different
- By implementation, a process can only access its own memory unless ..

Free Frame allocation



Before allocation

After allocation

Colorado State University

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

One page-table For each process

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**)

TLB: cache for page Table

Caching: Concept

- Widely used concept:
  - keep small subset of information likely to needed in near future in a fast accessible place

Examples:

- Cache Memory (“Cache”):
  Cache for Main memory
- Browser cache: for browser
- Disk cache
- Cache for Page Table: TLB

Challenges:

- Is the information in cache? Where?
- Hit rate vs cache size

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

TLB: cache for page Table

Associative Memory

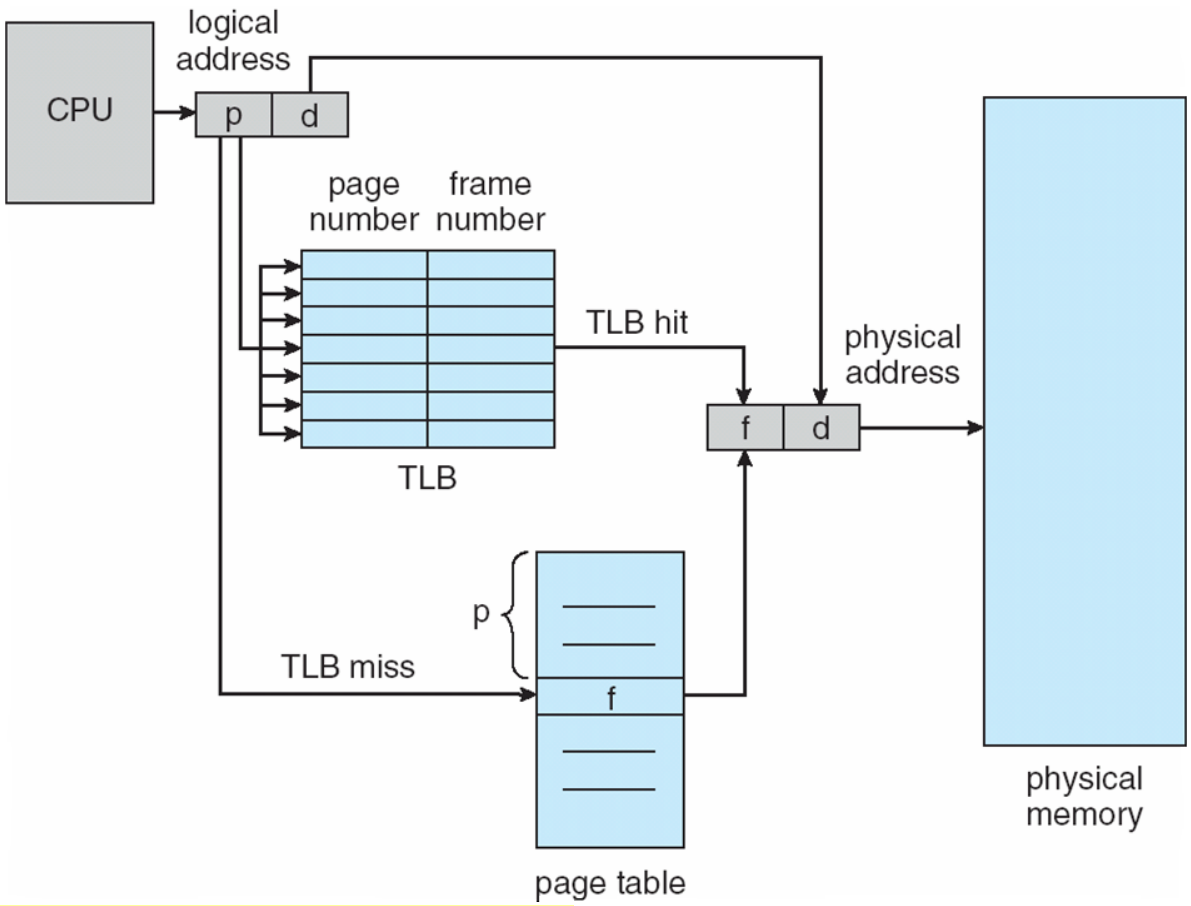
- Associative memory – parallel search

Page #	Frame #

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

Colorado State University

Paging Hardware With TLB



TLB Miss: page table access may be done using hardware or software

Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time** (**EAT**): probability weighted

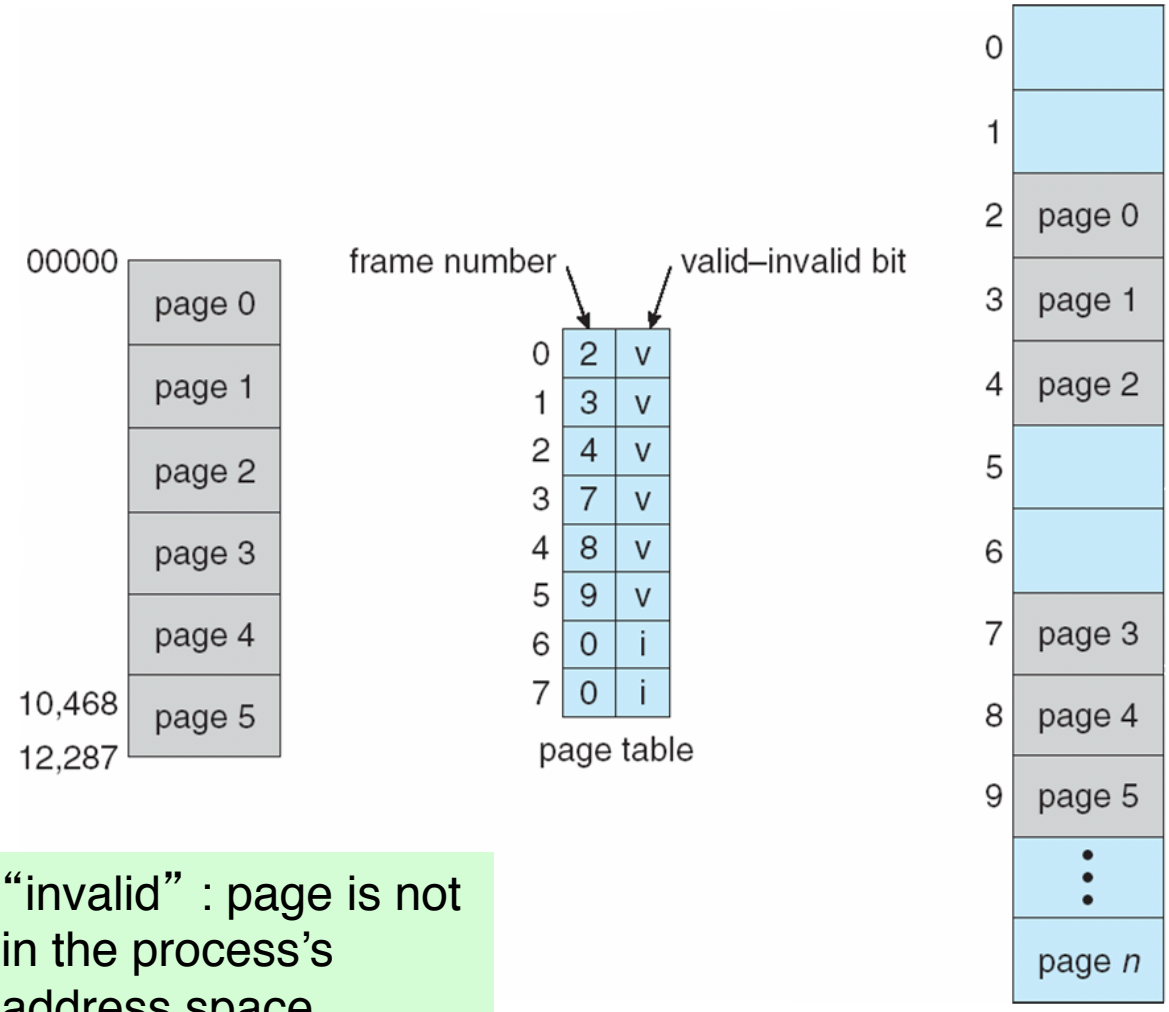
  $$EAT = (100 + \varepsilon)\ \alpha + (200+\varepsilon)(1 - \alpha)$$
- Ex:

  Consider α = 80%, ε = negligible for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%,
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

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
- Any violations result in a trap to the kernel

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

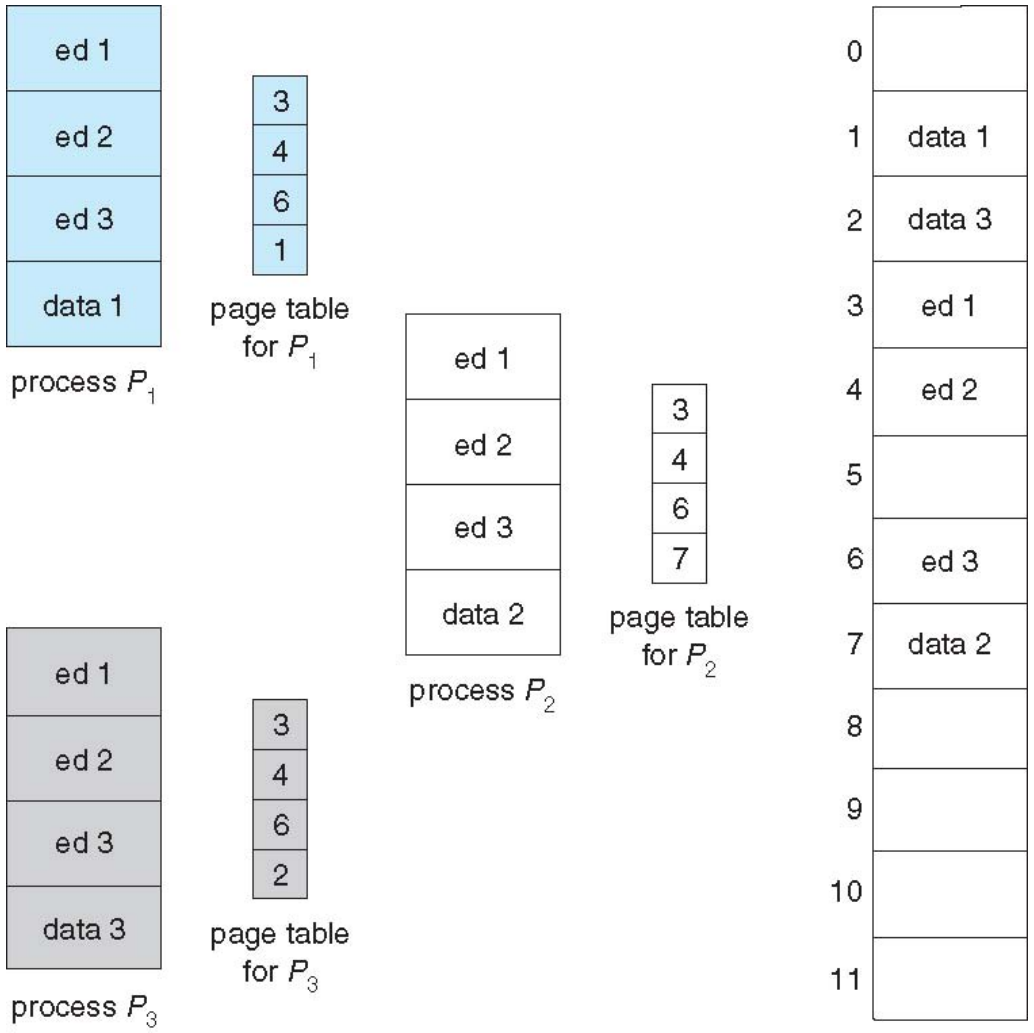


"invalid" : page is not in the process's address space.

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant** non-self modifying) 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



ed1, ed2, ed3
(3, 4, 6) shared

Colorado State University

Overheads in paging: Page table and internal fragmentation

Optimal Page Size: page table size vs int fragmentation tradeoff

- Average process size = ***s***
- Page size = ***p***
- Size of each page entry = ***e***
  - Pages per process = ***s/p***
  - ***se/p:*** Total page table space
- Total Overhead = Page table overhead + Internal fragmentation loss

  = se/p + p/2

Optimal Page size: Page table and internal fragmentation

- Total Overhead = $se/p + p/\mathbf{2}$
- Optimal: First derivative with respect to ***p,*** equate to 0

  $-se/p2 + 1/2 = 0$
- i.e. $p^2 = 2se$ or $p = (2se)^{0.5}$

Assume ***s*** = 128KB and ***e=8*** bytes per entry

- Optimal page size = 1448 bytes
  - In practice we will never use 1448 bytes
  - Instead, either 1K or 2K would be used
    - **Why?** Pages sizes are in powers of 2 i.e. $\mathbf{2^x}$
    - Deriving offsets and page numbers is also easier

Page Table Size

- 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}) entries
  - 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

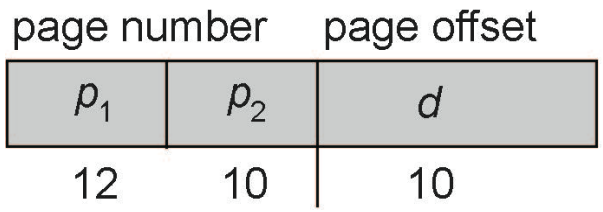
2^{10}	**1024 or 1 kibibyte**
2^{20}	1M mebibyte
2^{30}	1G gigibyte

Issues with large page tables

- Cannot allocate page table **contiguously** in memory
- Solutions:
  - Divide the page table into smaller pieces
  - **Page the page-table**
    - Hierarchical Paging

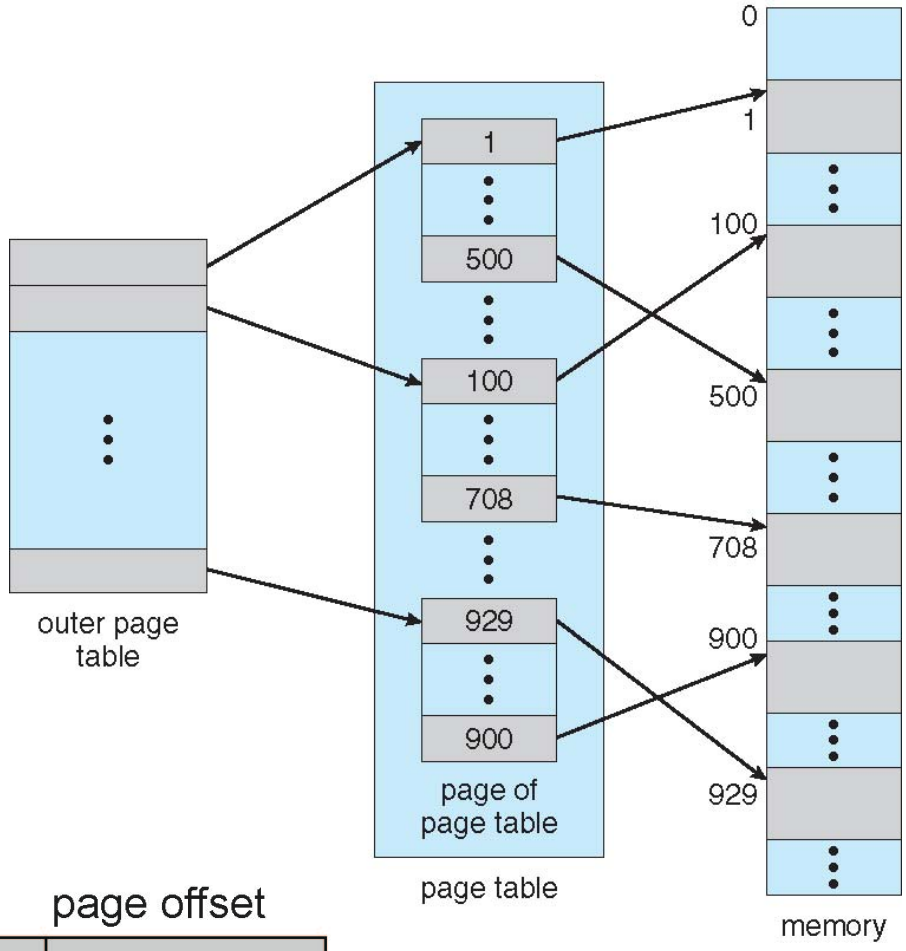
Hierarchical Page Tables

- 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



P1: indexes the outer page table
P2: page table: maps to frame

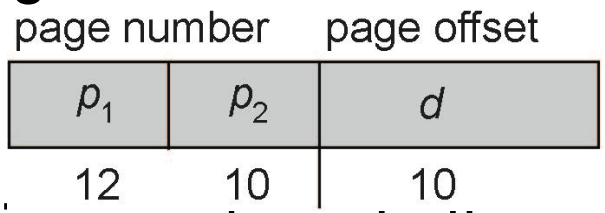
Two-Level Page-Table Scheme



page number		page offset
p_1	p_2	d
12	10	10

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**