# CS370 Operating Systems

**Colorado State University**

**Yashwant K Malaiya**

**Fall 22 L18**

**Main Memory**

**Slides based on**

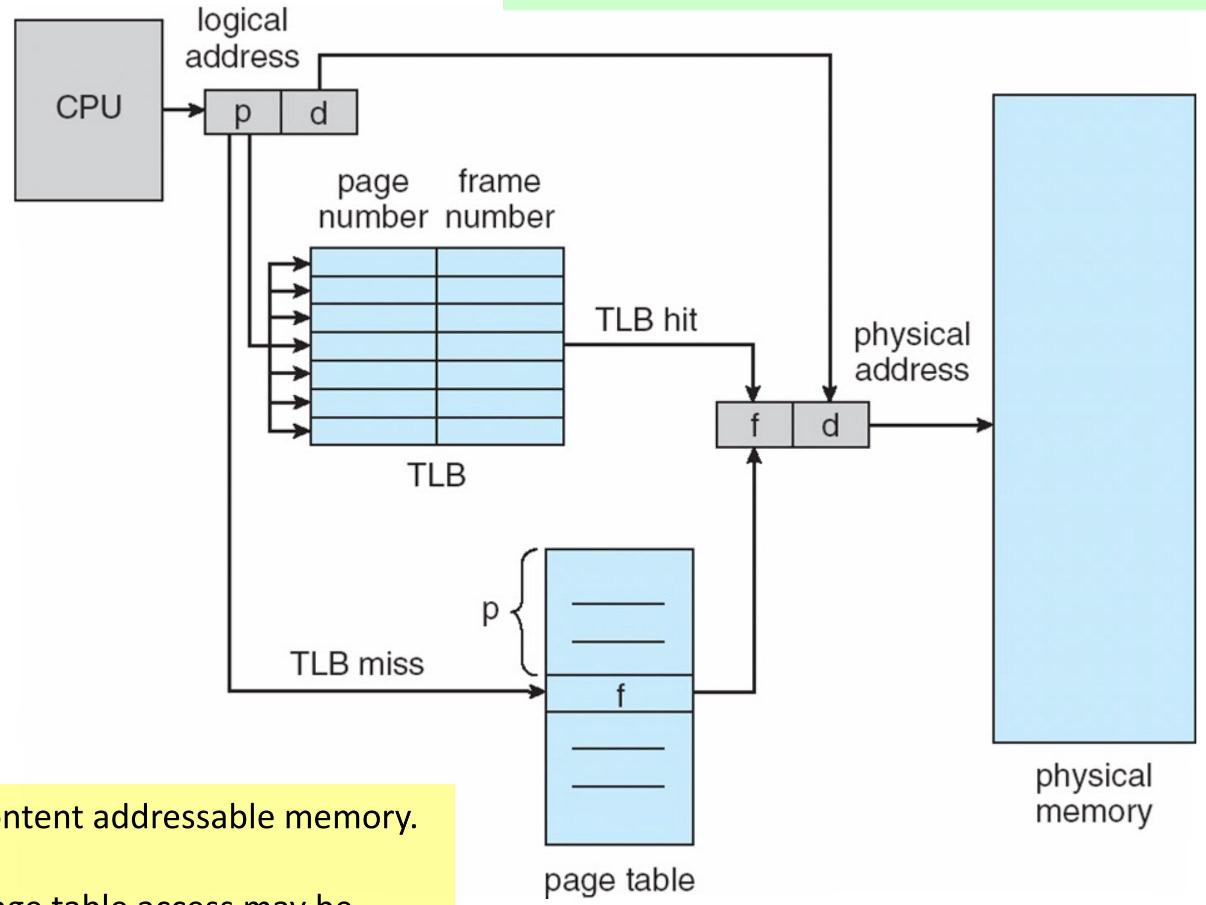
- Text by Silberschatz, Galvin, Gagne
- Various sources

## FAQ

- Is there is specific formula for calculating the physical address from the logical address? Page number to frame number lookup
- Each process has its own page table? Can there be a conflict in sharing physical memory? No, unless..
- Can the page table dynamically change?
- Where is the page table? Memory, with a part cached in TLB
- How to find the page table in memory? Page table base register
- Where is the TLB ? On the same chip as CPU.
- Why use associative memory for TLBs? To see if the mapping for a specific page is there.

# Paging Hardware With TLB

Page number p to frame number f



TLB: uses content addressable memory.

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

Colorado State University

# Effective Access Time

**General approach:** expected access time

Effective access time

$= \Pr\{\text{access type A}\}.\ \text{Access-time}_A +$

$\Pr\{\text{access type B}\}.\ \text{Access-time}_B$

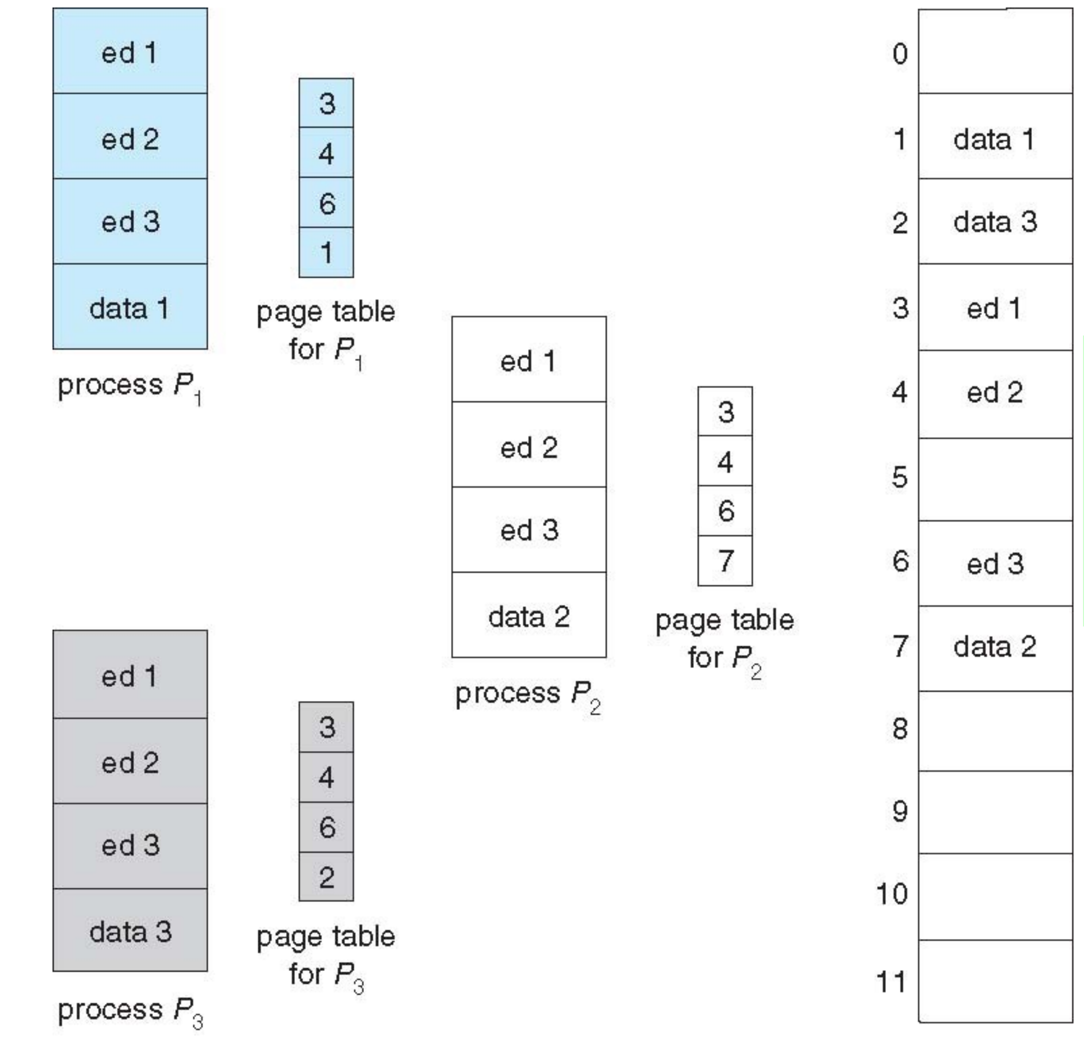
**Ex: effective access time with TLB/page table:**

- Associative Lookup = $\varepsilon$ time units
- Hit ratio = $\alpha$
- **Effective Access Time** (**EAT**): probability weighted

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

  Consider $\alpha$ = 80%, $\varepsilon$ = negligible for TLB search, 100ns for memory access
  - EAT = 100x0.80 + 200x0.20 = 120ns

# Shared Pages Example: 3 Processes



How are "pages" shared? Include in address space of both processes.

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 internal fragmentation tradeoff

- Average process size = ***s***
- Page size = ***p***
- Size of each entry in page table = ***e***

- Total Overhead = Page table overhead + Internal fragmentation loss

  $= se/p + p/2$

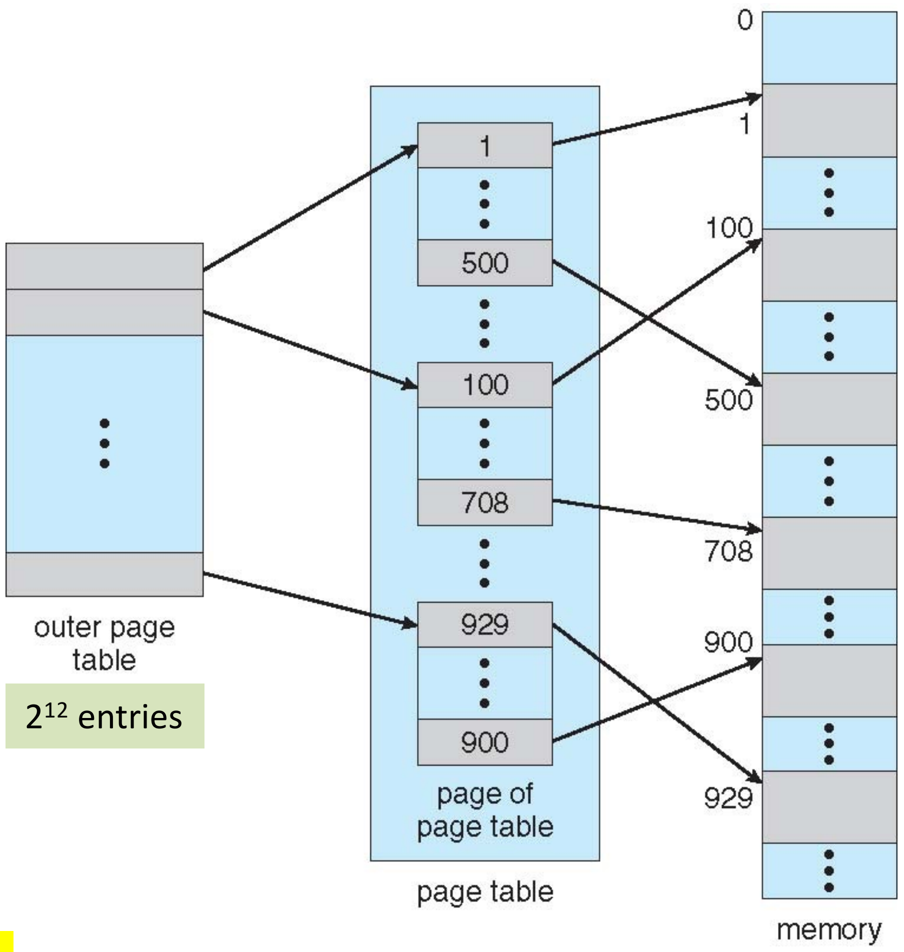
- *Optimal page size* $p = (2se)^{0.5}$

Colorado State University

## Issues with large page tables

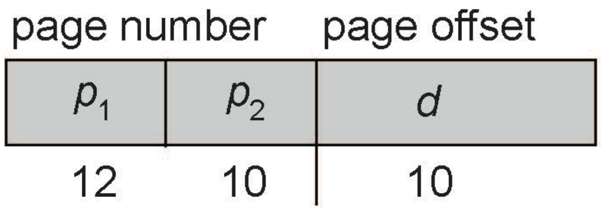
- Cannot allocate a large page table **contiguously** in memory
- Solution:
  - Divide the page table into smaller pieces
  - **Page the page-table**
    - Hierarchical Paging

# Two-Level Page-Table Scheme



$2^{12}$ entries

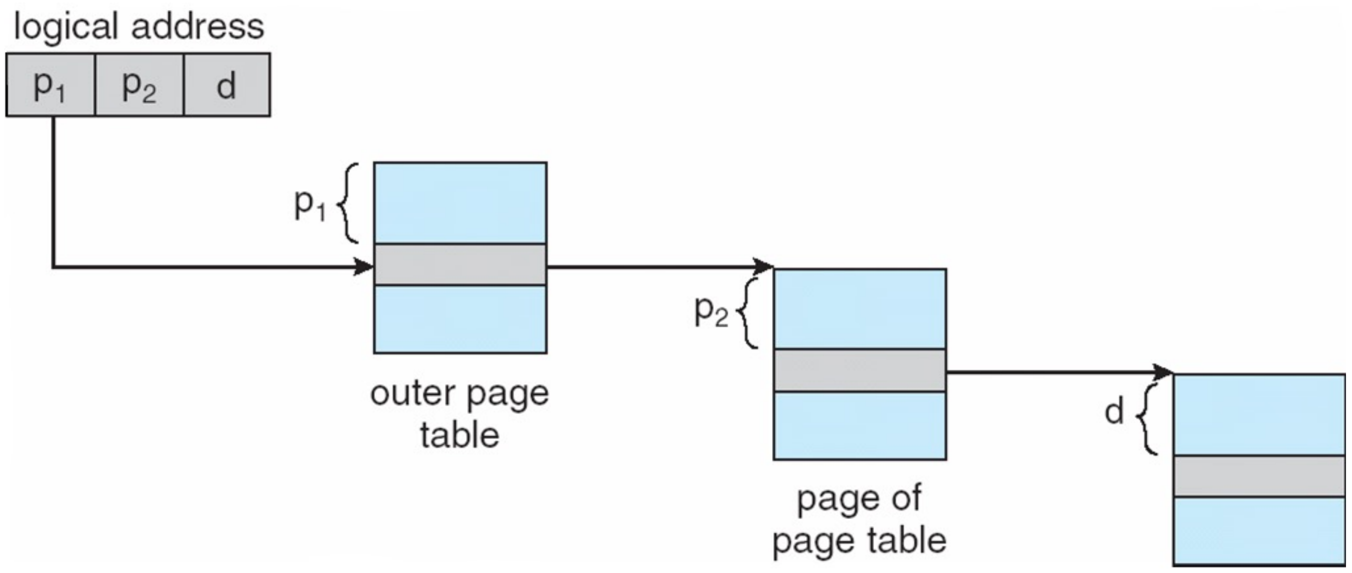
$2^{12}$ pages,
each with $2^{10}$ entries



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

Colorado State University

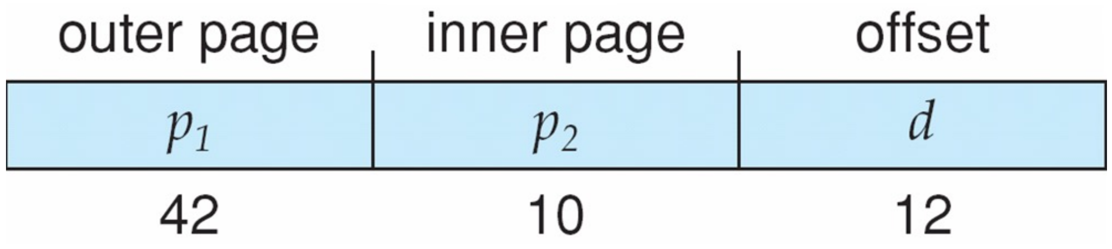
# Hierarchical Paging



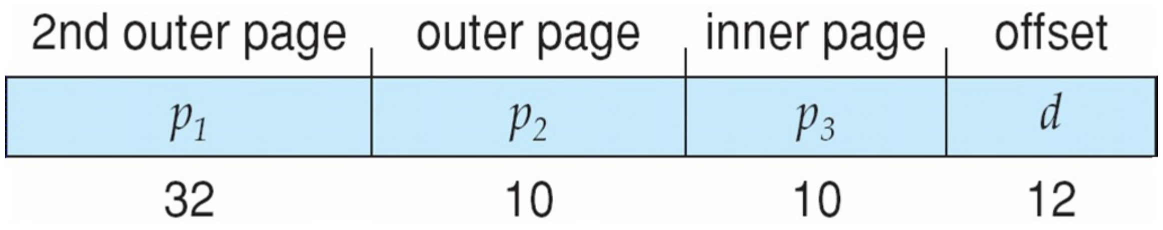
If there is a hit in the TLB (say 95% of the time), then average access time will be close to slightly more than one memory access time.

Colorado State University

## 64-bit add. Space: Three-level Paging Scheme



- Problem: Outer page table has $2^{42}$ entries!
- Approach: Divide the outer page table into 2 levels
  - 4 memory accesses!

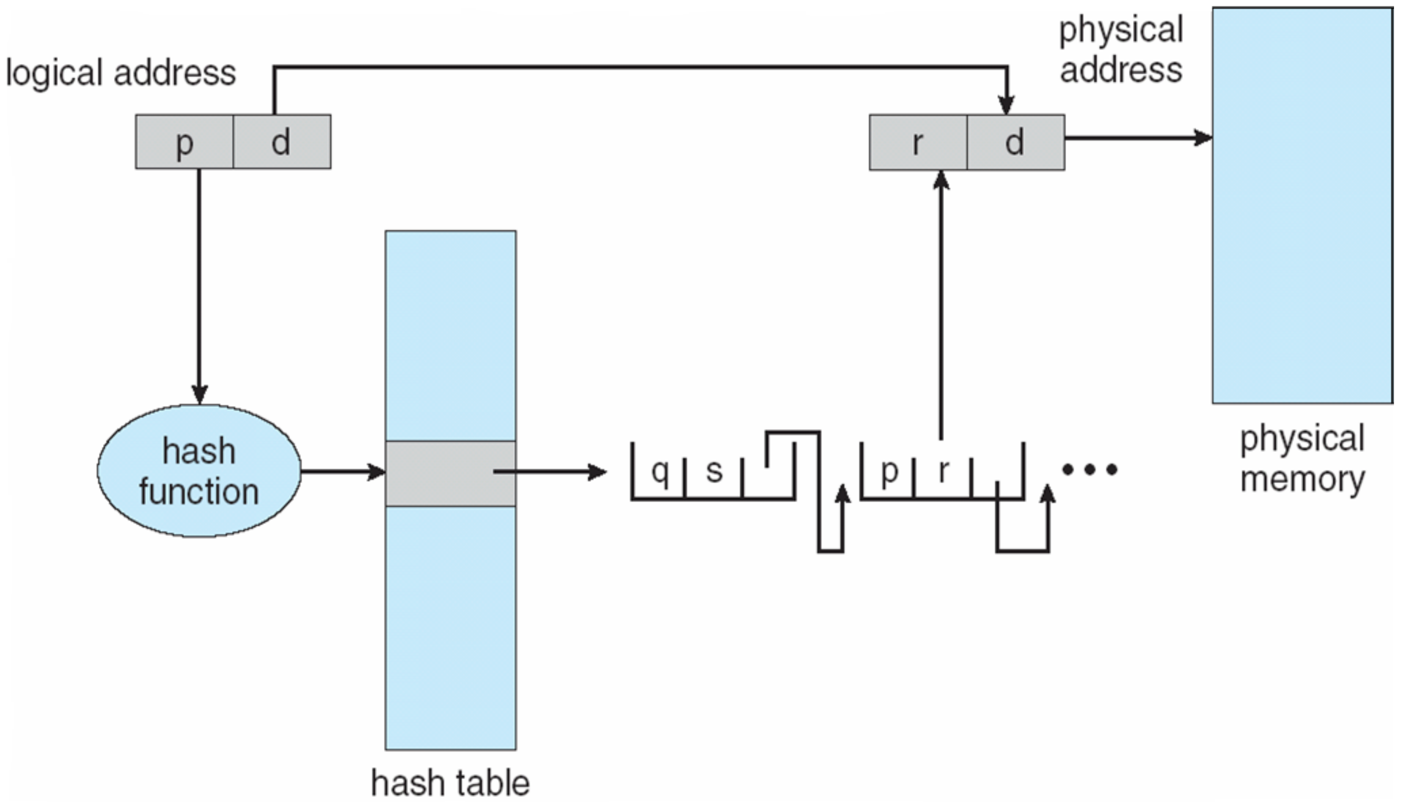


Colorado State University

# Hashed Page Tables

- 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)

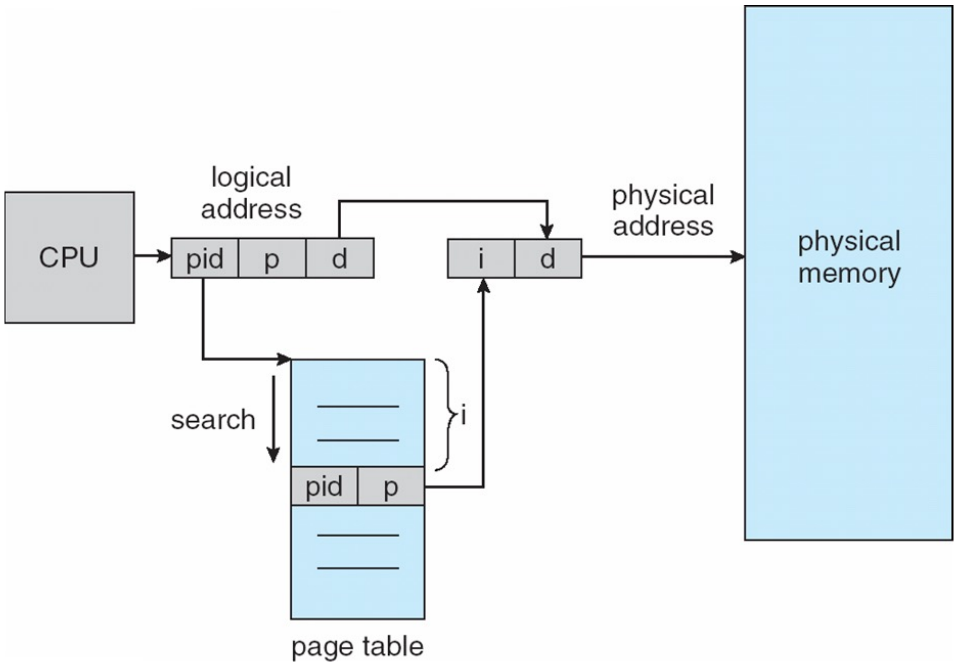
# Hashed Page Table



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

# 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 (“frame”)
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page



Search for pid, p, offset i is the physical frame address
Note: multiple processes in memory

Colorado State University

# Inverted Page Table

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address. Not possible.

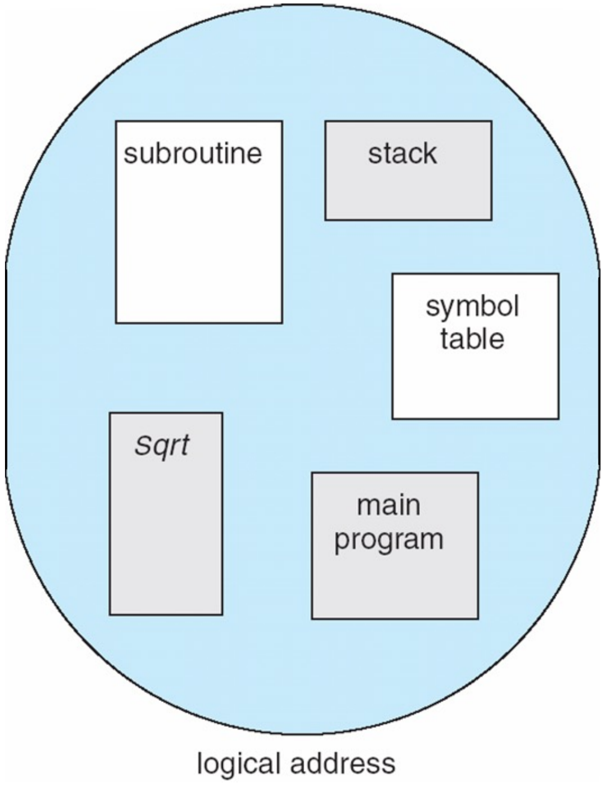
Used in IA-64 ..

Colorado State University

# Segmentation Approach

Memory-management scheme that supports user view of memory

- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure, function, method
    - object
    - local variables, global variables
    - common block
    - stack, arrays, symbol table
- Segment table
  - Segment-table base register (STBR)
  - Segment-table length register (STLR)
- segments vary in length, can very dynamically
- Segments may be paged
- Used for x86-32 bit
- Origin of term "segmentation fault"

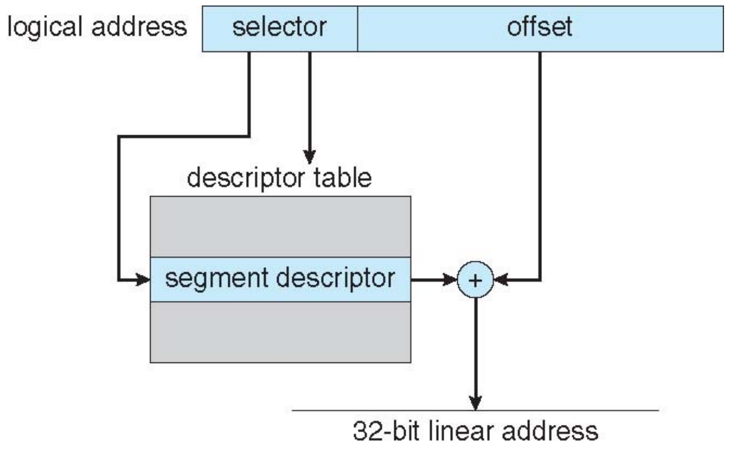


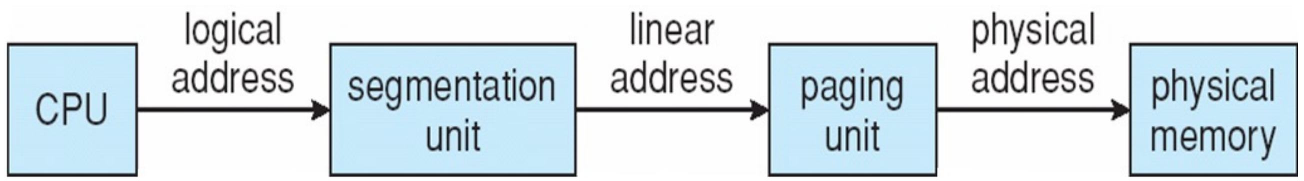
Colorado State University

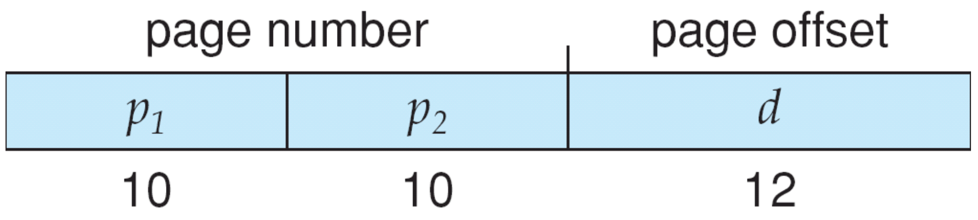
## Examples

- Intel IA-32 (x386-Pentium)
- x86-64 (AMD,  Intel)
- ARM (Acorn > ARM Ltd > Softbank > Nvidea)

## Logical to Physical Address Translation in IA-32

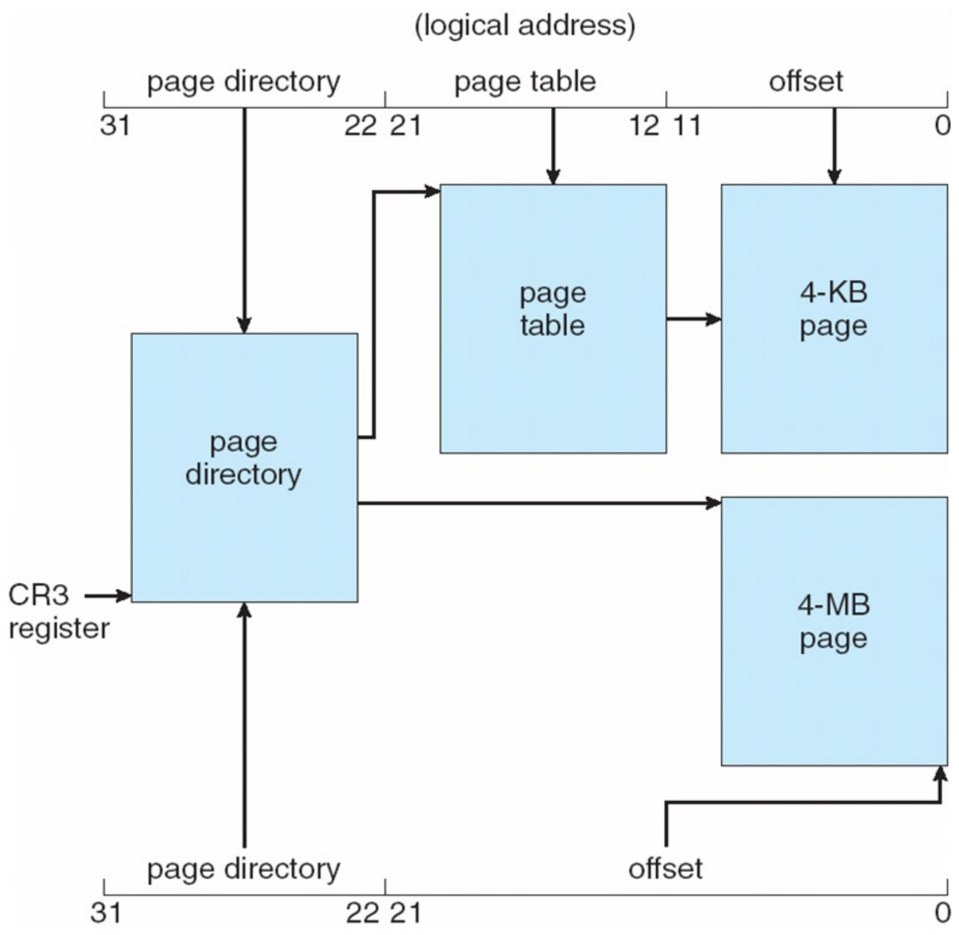






Colorado State University

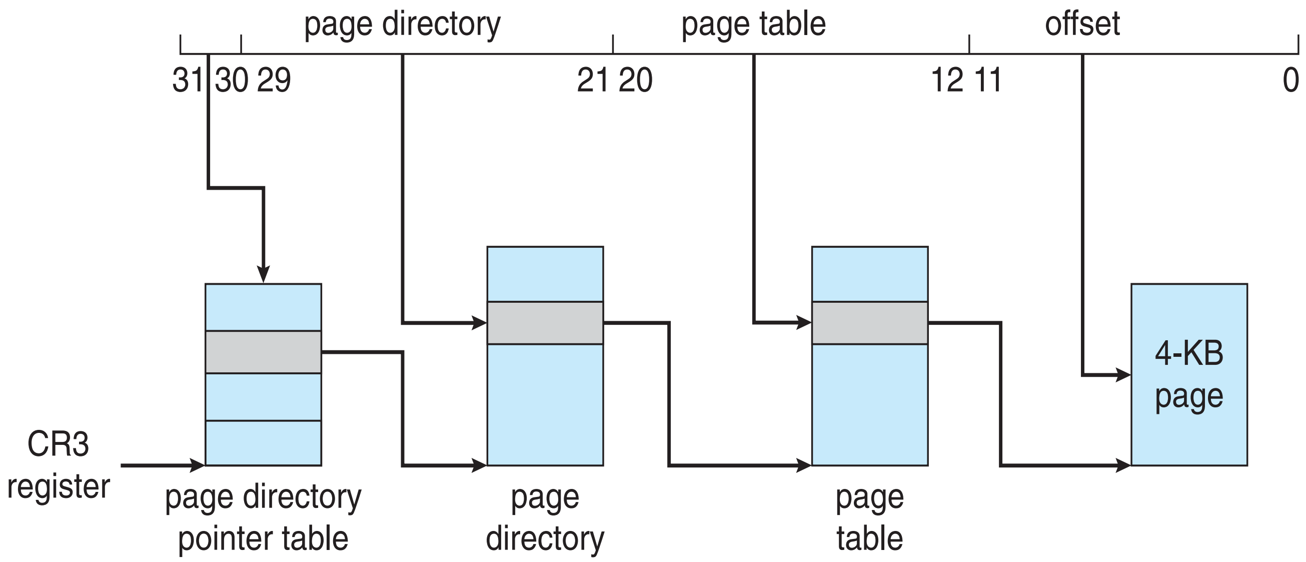
# Intel IA-32 Paging Architecture



Colorado State University

# Intel IA-32 Page Address Extensions

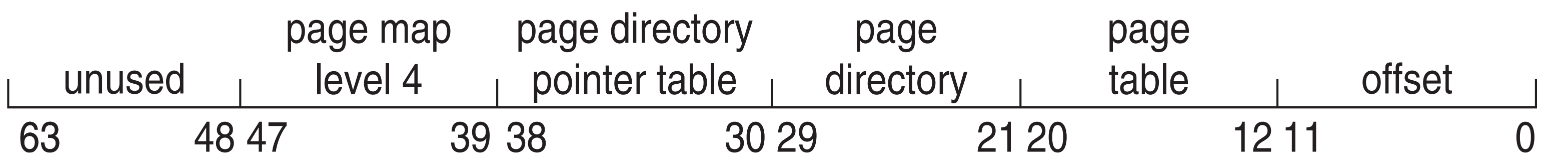
- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space by increasing  frame address bits.



Colorado State University

# Intel x86-64

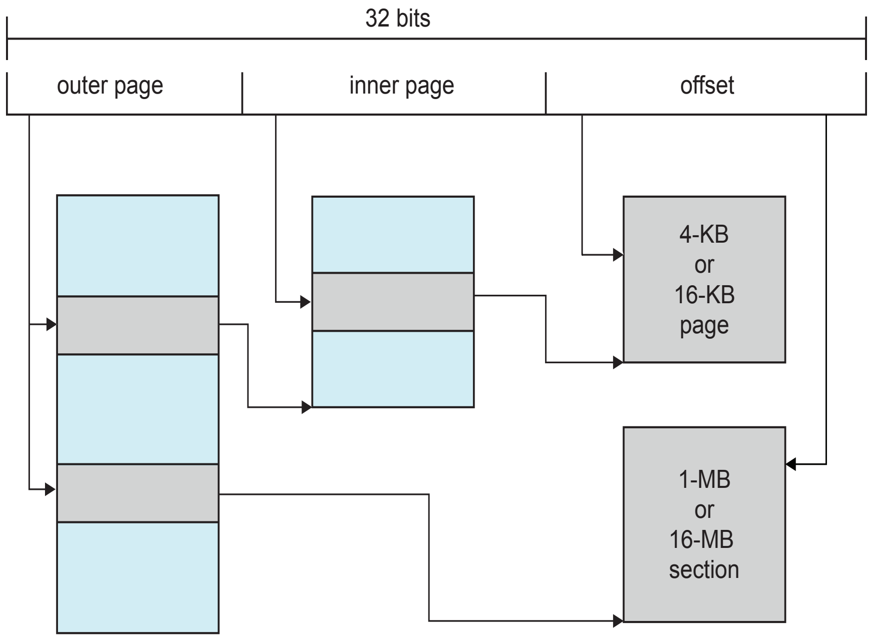
- Intel x86 architecture based on AMD 64 bit architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing or perhaps 52
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PageAddressExtensions so virtual addresses are 48 bits and physical addresses are 52 bits



Exabyte: $1024^6$ bytes

# Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



Colorado State University

# CS370 Operating Systems

## Colorado State University

## Yashwant K Malaiya

## Spring 2022

## Virtual Memory

**Slides based on**

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

## What we expect in future

iClicker Exit Poll question

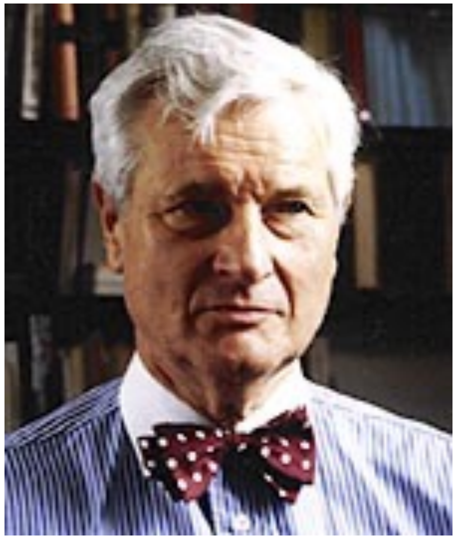
What major tech achievements are you guys looking forward to in the next decade?

# Virtual Memory: Objectives

Stephanie © 2007
www.onlinedatingmagazine.com
"You say we went out and I never called? I can't remember. My virtual memory must be low!"

- A virtual memory system
- Demand paging, page-replacement algorithms, allocation of page frames to processes
- Threshing, the working-set model
- Memory-mapped files and shared memory and
- Kernel memory allocation

# Fritz-Rudolf Güntsch: Virtual Memory



Fritz-Rudolf Güntsch (1925-2012) at the Technische Universität Berlin in 1956 in his doctoral thesis, *Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation*.

First used in Atlas, Manchester, 1962

PCs: Windows 95

When was Win 95 introduced?

## Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program uses less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

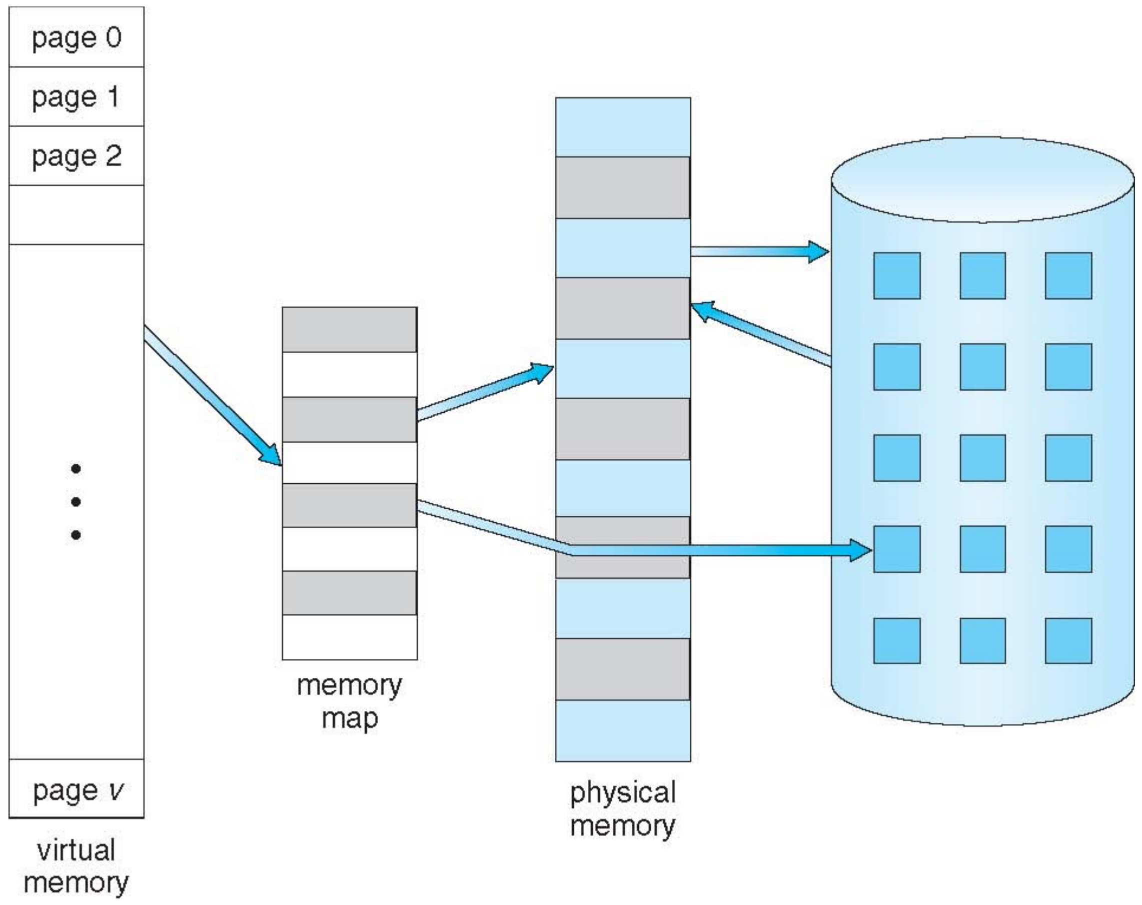
## Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory
- **Virtual address space** – logical view of how process views memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

That is the new idea

Colorado State University

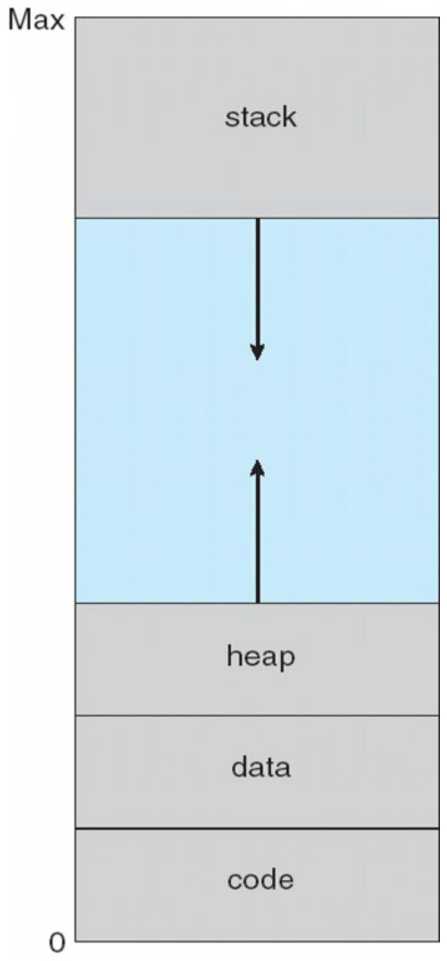
## Virtual Memory That is Larger Than Physical Memory



Colorado State University

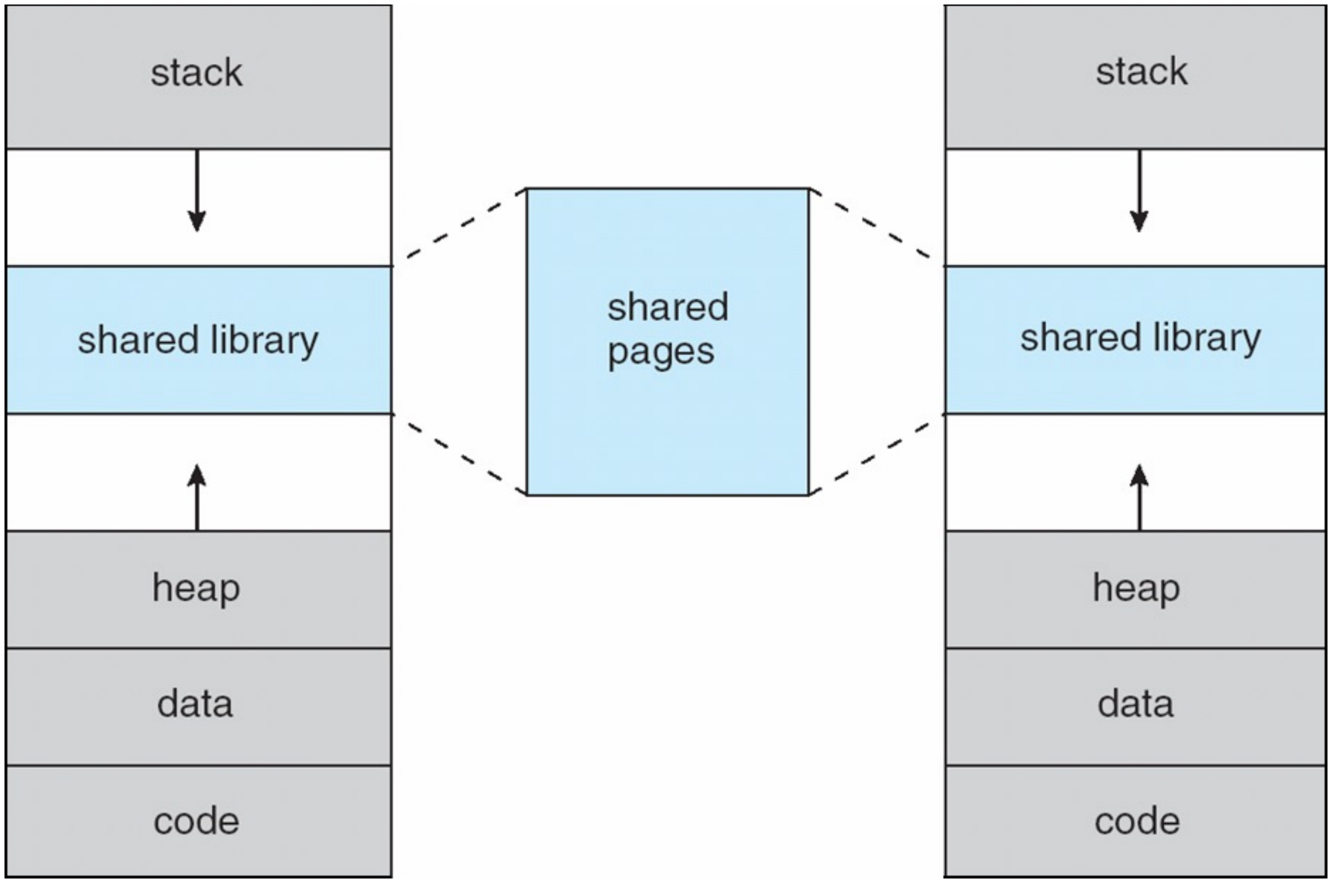
# Virtual-address Space: advantages

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Colorado State University

## Shared Library Using Virtual Memory



Colorado State University

# Demand Paging

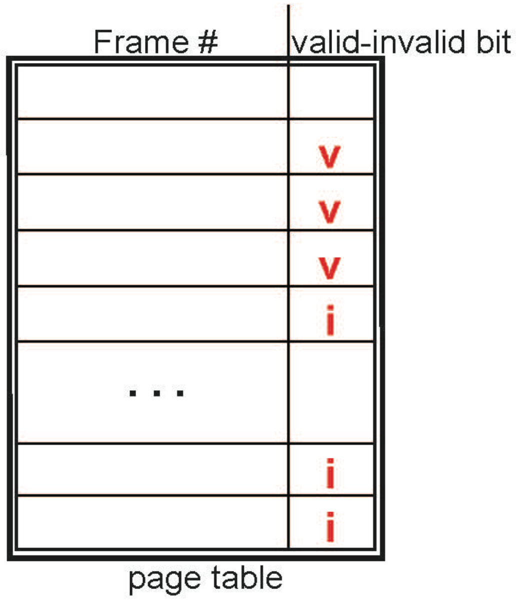
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed: **Demand paging**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **“Lazy swapper”** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

# Demand paging: Basic Concepts

- Demand paging: pager brings in only those pages into memory what are needed
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non-demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

# Valid-Invalid Bit

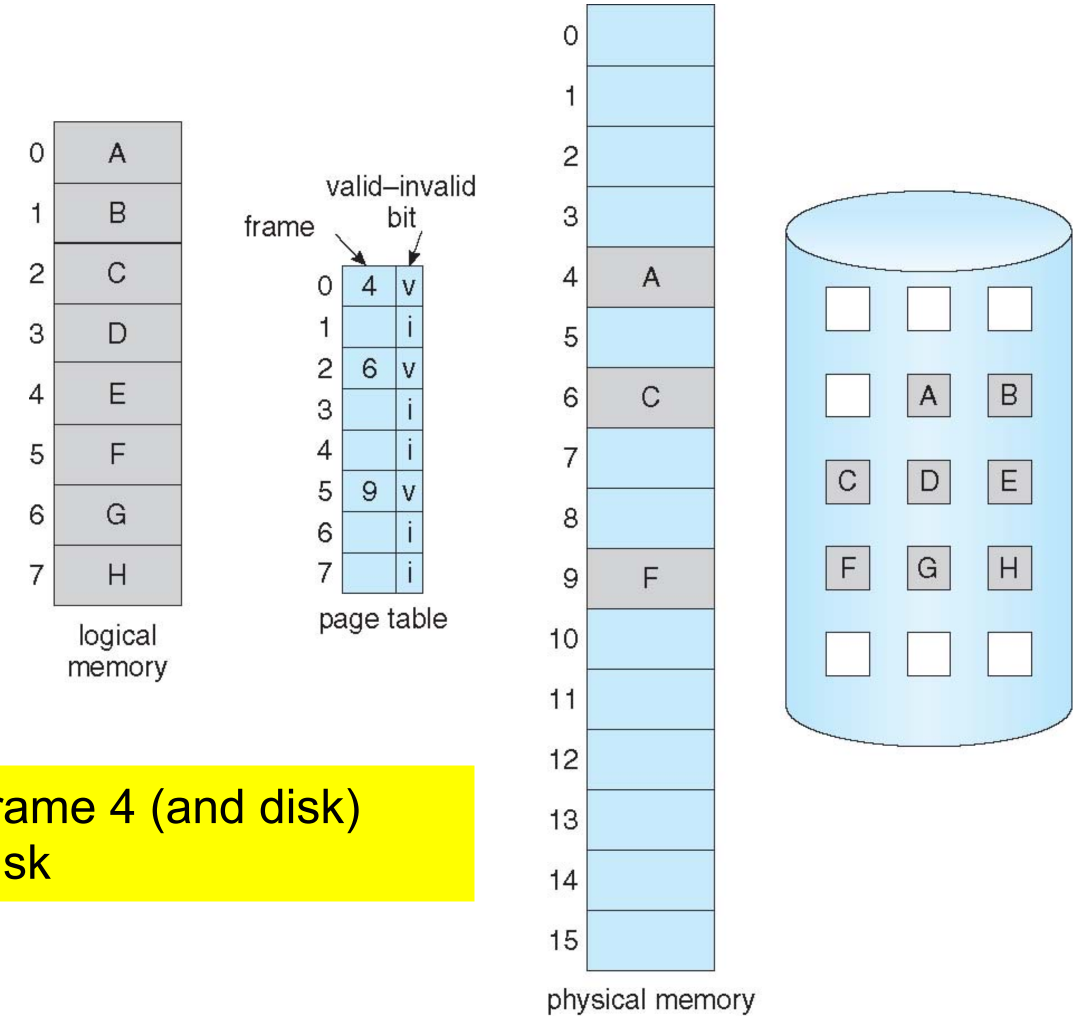
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Colorado State University

## Page Table When Some Pages Are Not in Main Memory



Page 0 in Frame 4 (and disk)
Page 1 in Disk

# Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: Page fault

**Page fault**

1. Operating system looks at a table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory, but in *backing storage*, ->2
2. Find free frame
3. Get page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Page fault: context switch because disk access is needed

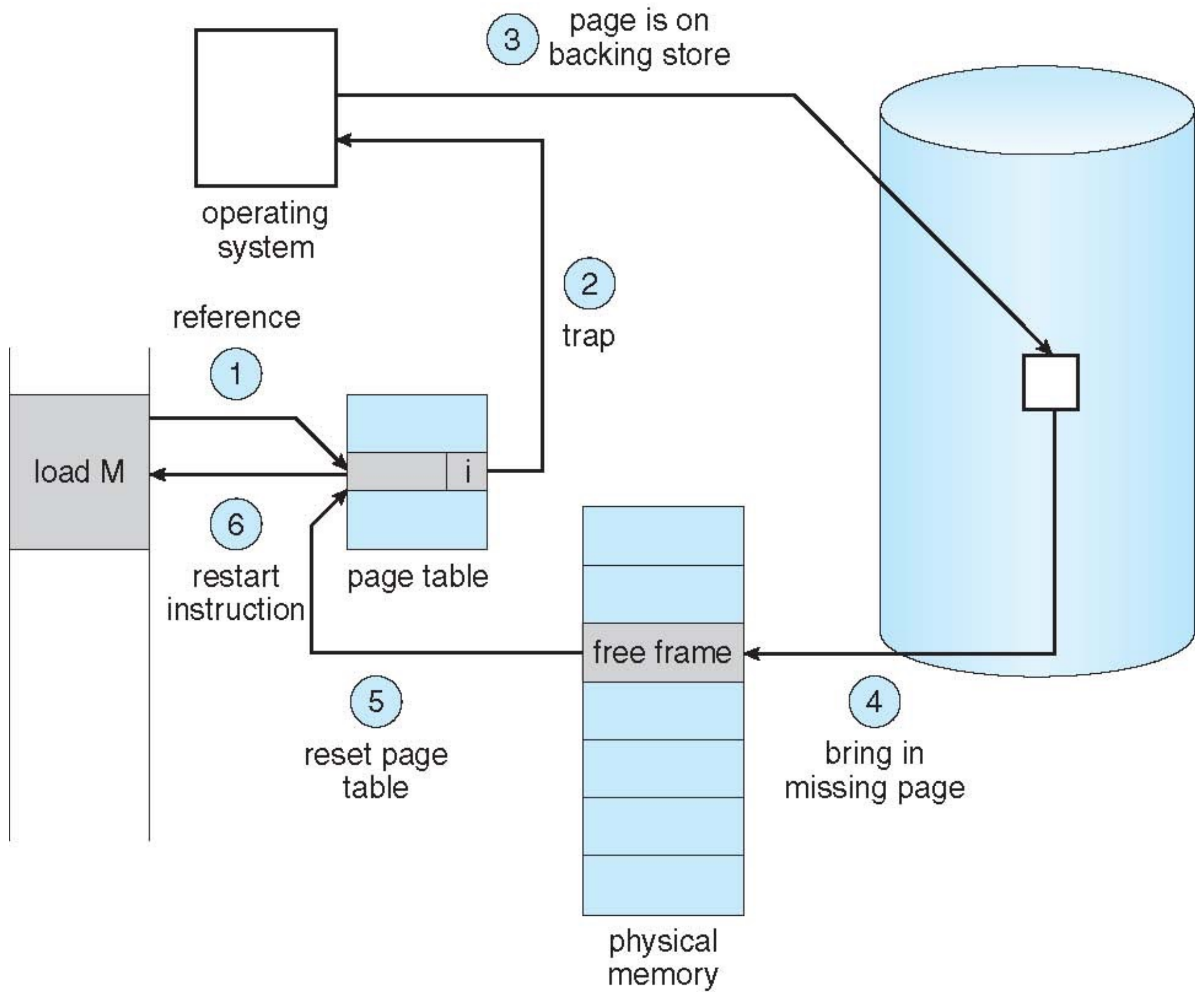
## Technical Perspective: Multiprogramming

Solving a problem gives rise to a new class of problem:

- Contiguous allocation. Problem: external fragmentation
- Non-contiguous, but entire process in memory: Problem: Memory occupied by stuff needed only occasionally. Low degree of Multiprogramming.
- Demand Paging: Problem: page faults
- How to minimize page faults?

Colorado State University

# Steps in Handling a Page Fault



Colorado State University

## Stages in Demand Paging (worse case)

1. **Trap to the operating system**
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. **Issue a read from the disk to a free frame:**
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. **While waiting, allocate the CPU to some other user**
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. **Correct the page table and other tables to show page is now in memory**
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then **resume the interrupted instruction**

# Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – relatively long time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

Hopefully p <<1

EAT = (1 – *p*) x memory access time
+ *p* (page fault overhead
+ swap page out + swap page in )

Page swap time = seek time + latency time

Colorado State University

## Demand Paging Simple Numerical Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 ns + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000 nanosec.
  = 200 + p x 7,999,800 ns

Linear with page fault rate

- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent, **p = ?**
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

We make some simplifying assumptions here.

Colorado State University

## Issues: Allocation of physical memory to I/O and programs

- Memory used for holding **program** pages
- **I/O buffers** also consume a big chunk of memory
- Choices:
  - Fixed percentage set aside for I/O buffers or
  - Processes and the I/O subsystem compete

# Demand paging and the limits of logical memory

- Without demand paging
  - All pages of process **must be** in physical memory
  - Logical memory **limited** to size of physical memory
- With demand paging
  - All pages of process **need not be** in physical memory
  - Size of logical address space is **no longer constrained** by physical memory
- Example
  - 40 pages of physical memory
  - 6 processes each of which is 10 pages in size
    - But each process only needs 5 pages *as of now*
  - Run 6 processes with 10 pages to spare

Higher degree of multiprogramming

Colorado State University