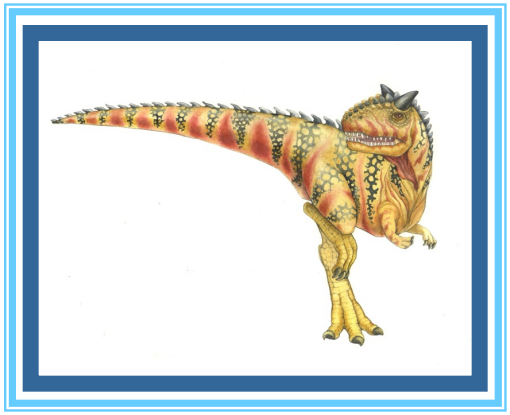
Lecture 8: Main Memory



Lecture 8: Memory Management

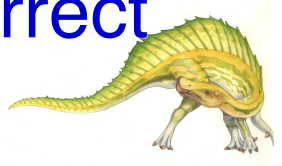
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of Page Table
- Example: Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Objectives

- To Provide a Detailed Description of Various Ways of Organizing Memory Hardware
- To Discuss Various Memory-Management Techniques, including Paging and Segmentation
- To Provide a Description of Intel Pentium, which Supports both Pure Segmentation and Segmentation with Paging

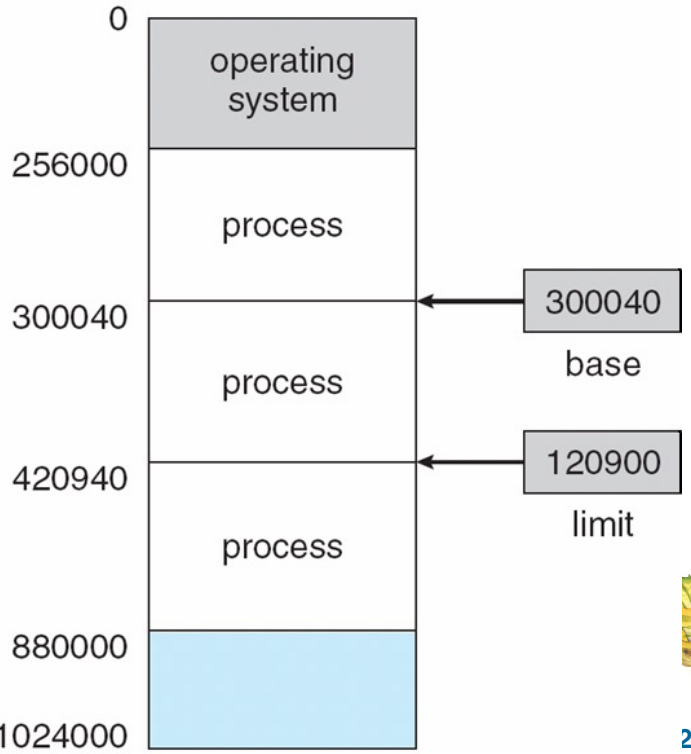
Background

- Program must be Brought (from disk) into Memory and Placed within a Process for it to be Run
- Main Memory (MM) and Registers are Only Storage CPU can Access Directly
- Memory Unit only Sees
  - A Stream of Addresses + Read requests, or Address + Data and Write Requests
- Register Access in one CPU Clock (or Less)
- MM can Take Many Cycles, Causing a **Stall**
- **Cache** Sits between MM and CPU registers
- Protection of Memory required to Ensure Correct Operation

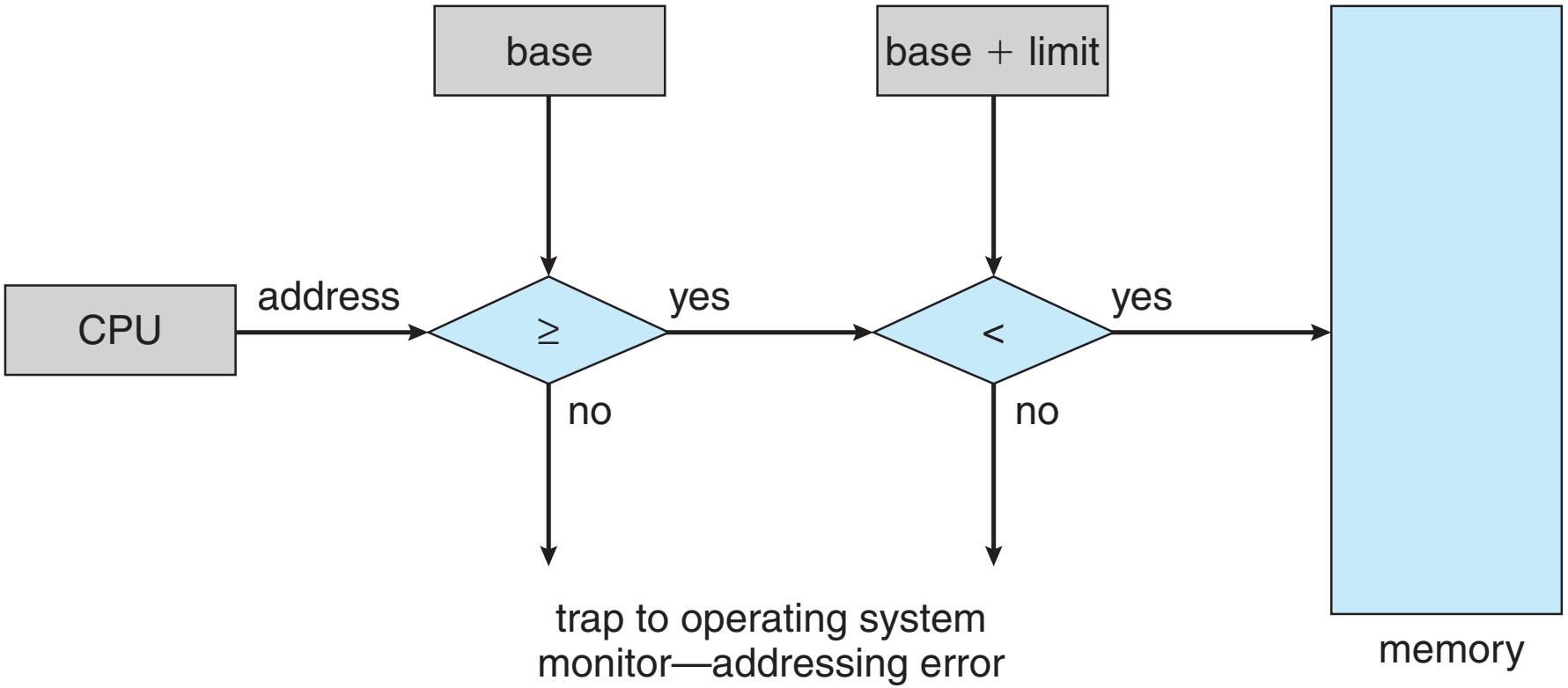


Base and Limit Registers

- A Pair of **Base** and **Limit Registers** Define Logical Address Space
- CPU must Check Every Memory Access
  - Generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Address Binding

- Programs on Disk, Ready to be Brought into Memory to Execute Form an **Input Queue**
  - Without support, must be loaded into addr: 0000
- Inconvenient to Have First User Process Physical Address Always at 0000
  - How can it not be?
- Further, Addresses Represented in Different Ways at Different Stages of a Program's Life
  - Source code, compiled code, linker/loader addresses

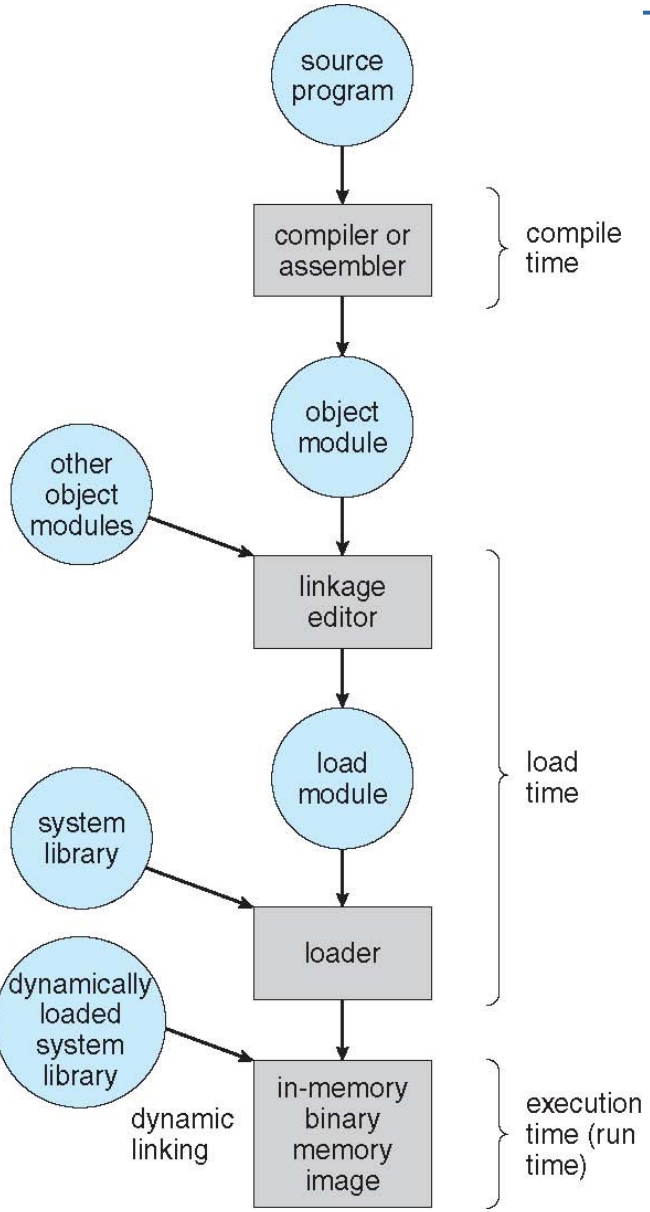
Address Binding (cont.)

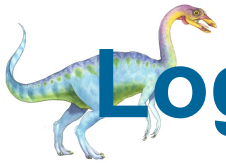
- Different Ways of Addresses Representation
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. “14 bytes from beginning of module”
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address Binding to Memory Addresses can Happen at Three Different Stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated
    - Must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if process can be moved during its execution from one memory segment to another
    - Need HW support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program





Logical vs. Physical Address Space

- Concept of a Logical Address Space
  - Central to Proper Memory Management
  - Bound to a Separate **Physical Address Space**
- **Logical address**
  - Generated by CPU
  - Aka, **virtual address**
- **Physical Address**
  - Address seen by memory unit
- Logical and Physical Addresses
  - Are same in compile-time and load-time address-binding schemes
  - Differ in execution-time address-binding scheme

Logical vs. Physical Address Space (cont.)

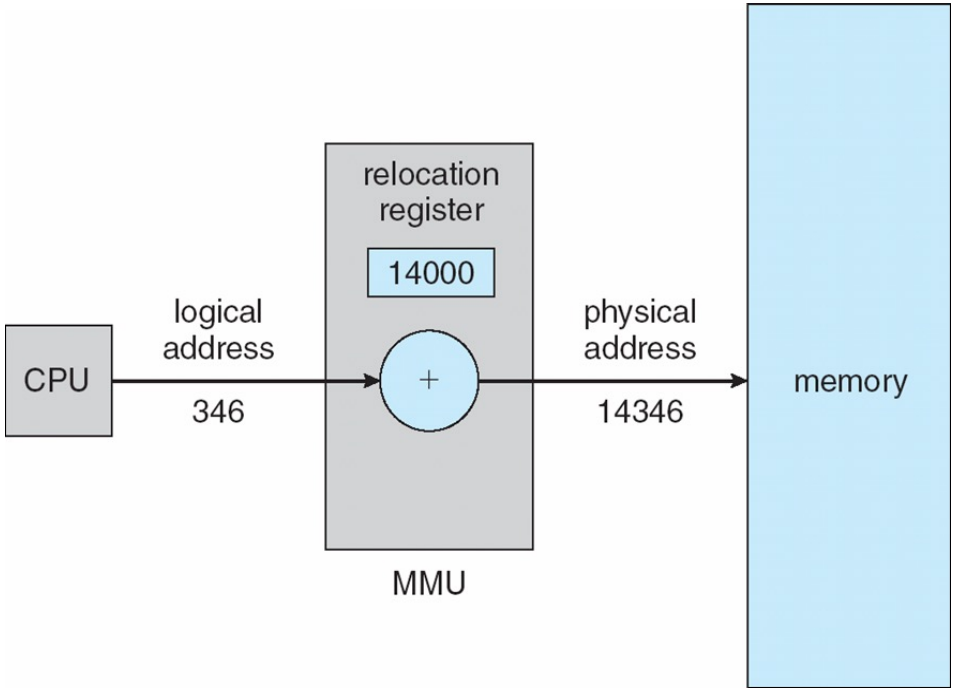
- **Logical Address Space**
  - Set of all logical addresses generated by a program
- **Physical Address Space**
  - Set of all physical addresses generated by a program

Logical vs. Physical Address Space (cont.)

- Memory Management Unit (MMU)
  - HW device that at run-time maps virtual to physical address
  - Many methods possible
    - Will be discussed in this lecture
- To start, Consider Simple Scheme
  - Value in base register is added to every address generated by a user process at time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

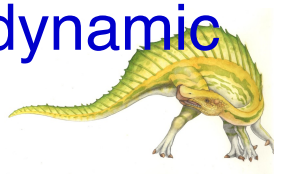
Logical vs. Physical Address Space (cont.)

- User Program Deals with *Logical* Addresses
  - It never sees *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses



Dynamic Relocation using a Relocation Register

- Routine not Loaded until it is Called
- Better Memory-Space Utilization
  - Unused routine never loaded
- All Routines Kept on Disk in Relocatable Load Format
- Useful when Large Amounts of Code are Needed to Handle Infrequently Occurring Cases
- No Special Support from OS required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

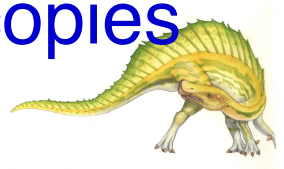
- **Static Linking** – system libraries and program code combined by loader into binary program image
  - Wastes both disk space and main memory
- Dynamic Linking – linking postponed until execution time
  - Small piece of code, **Stub**, used to locate appropriate memory-resident library routine
  - Stub replaces itself with address of routine, and executes routine

Dynamic Linking (cont.)

- OS Checks if Routine is in Memory
  - If not in address space, add to address space
- Dynamic Linking is Particularly Useful for Libraries
  - Processes that use a language library execute only one copy of library code
  - Libraries updates will be automatically applied

Swapping

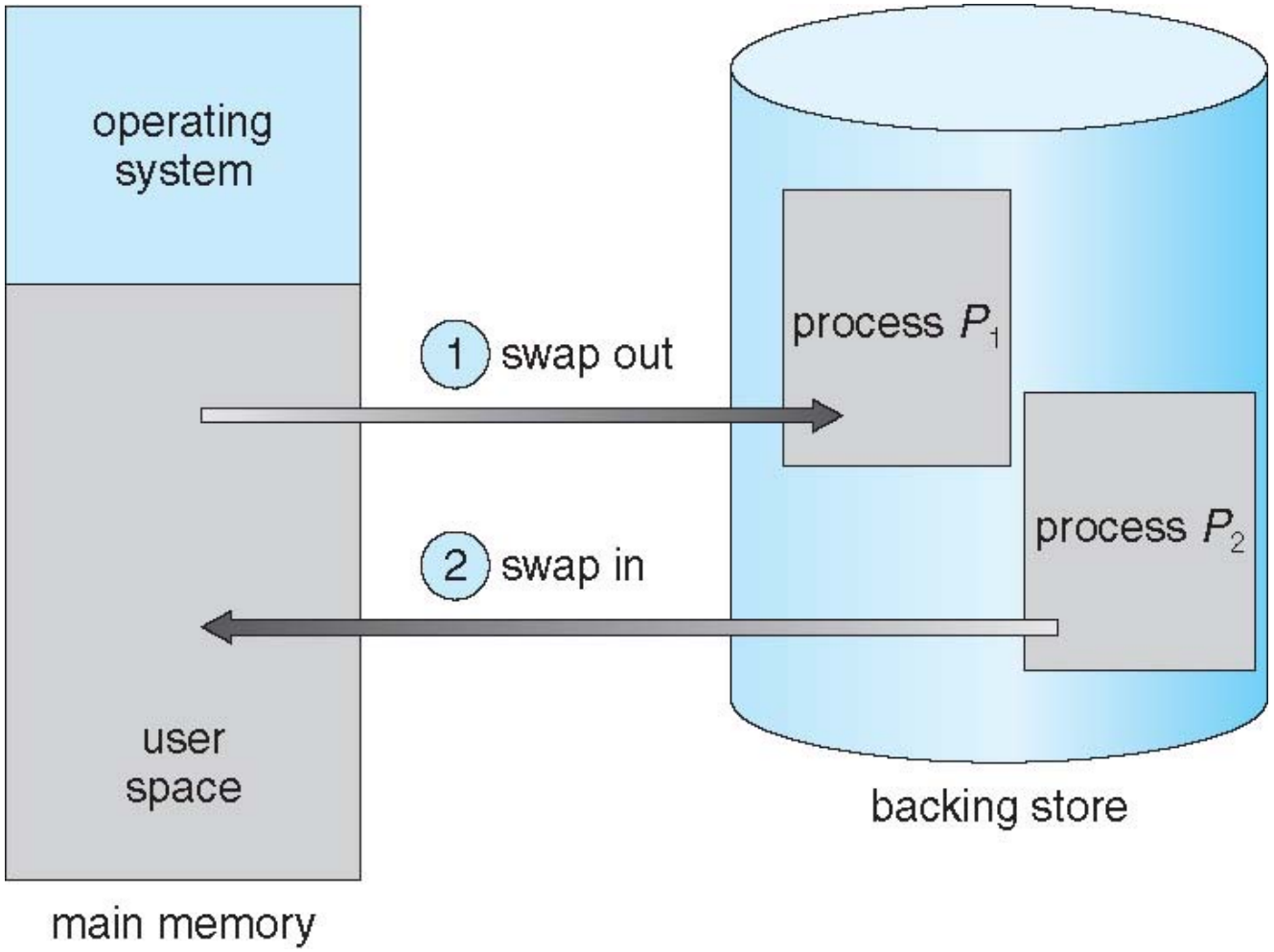
- A Process can be **Swapped** Temporarily out of Memory to a Backing Store, and Brought back into Memory for Continued Execution
- Why Swap out?
  - Quantum of round robin expired ➔ need to bring a new process ➔ not enough memory space
    - Total physical memory space of processes can exceed physical memory
- **Backing store**
  - Fast disk large enough to accommodate copies of all memory images for all users



Swapping (cont.)

- Major part of Swap Time is Transfer Time
  - Total transfer time is directly proportional to amount of memory swapped
- System Maintains a **Ready Queue** of ready-to-run processes
  - Which have memory images on disk (or in main memory)

Schematic View of Swapping

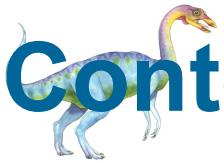


Context Switch Time and Swapping

- Next Processes to be Put on CPU Not in MM
  - Need to swap out a process & swap in target process
- Context Switch Time can then be Very High
- Example: 100MB Process
  - Swapping to HDD with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)

Swapping (cont.)

- Q: Does Swapped out Process Need to Swap Back into Same Physical Addresses?
- Ans: Depends on address binding method
- Standard Swapping is Too Time-Consuming
- ➔ Modified Versions of Swapping 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



Context Switch Time and Swapping (cont.)

- Can reduce Context Switch Time
  - If reduce size of memory swapped
    - by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
- Other Constraints as well on Swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead

Memory Management Schemes

- Main Memory must support OS + User Processes
  - Limited Resource ➔ must allocate efficiently
- Memory Management Schemes
  - Contiguous Allocation
  - Segmentation
  - Paging
  - Segmentation + Paging

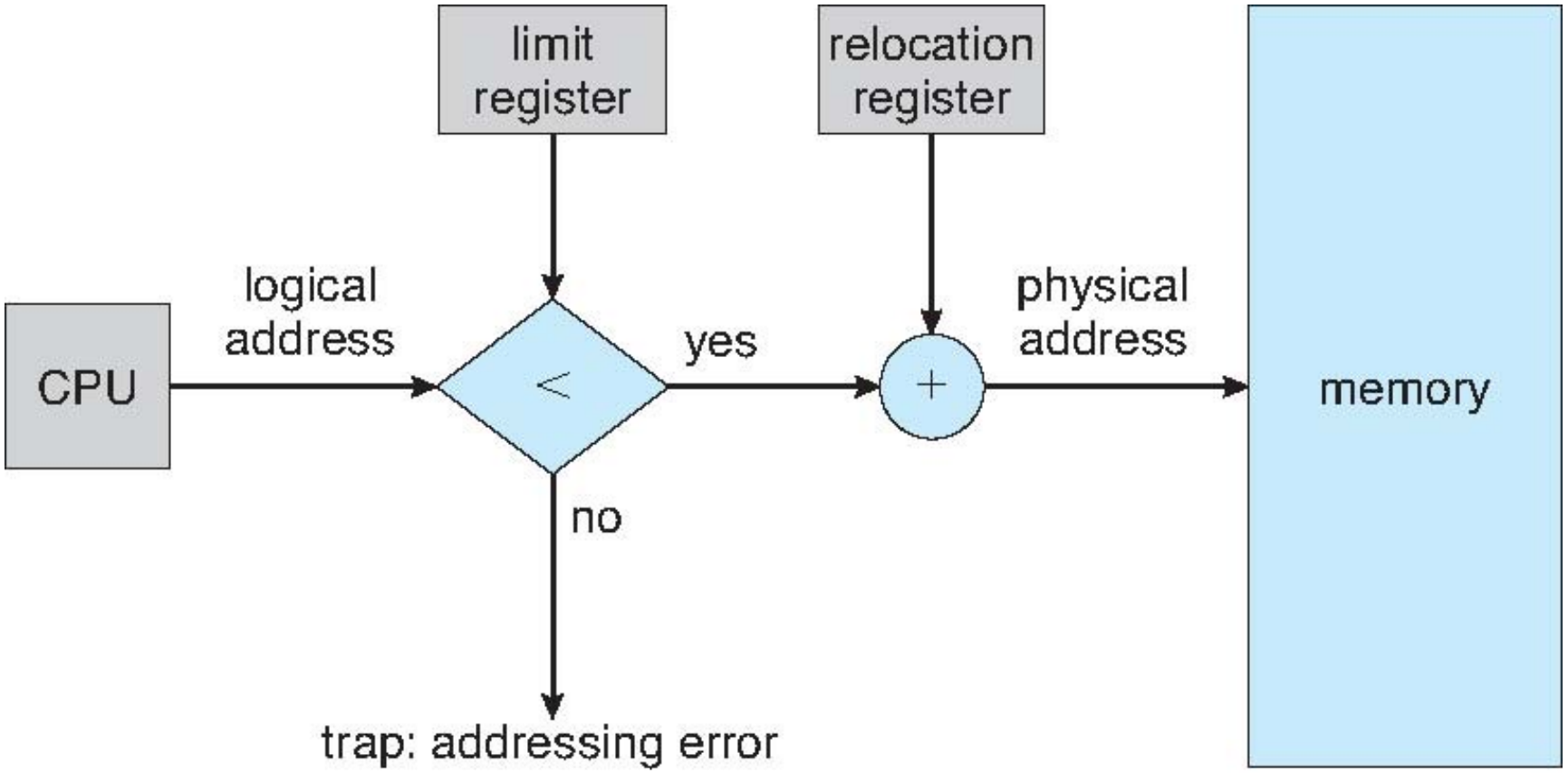
Contiguous Allocation

- Contiguous Allocation is one Early Method
- Main Memory Usually into two **partitions**:
  - Resident OS, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (cont.)

- Relocation Registers used to Protect User Processes from each other, and from Changing OS code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

HW Support for Relocation and Limit Registers



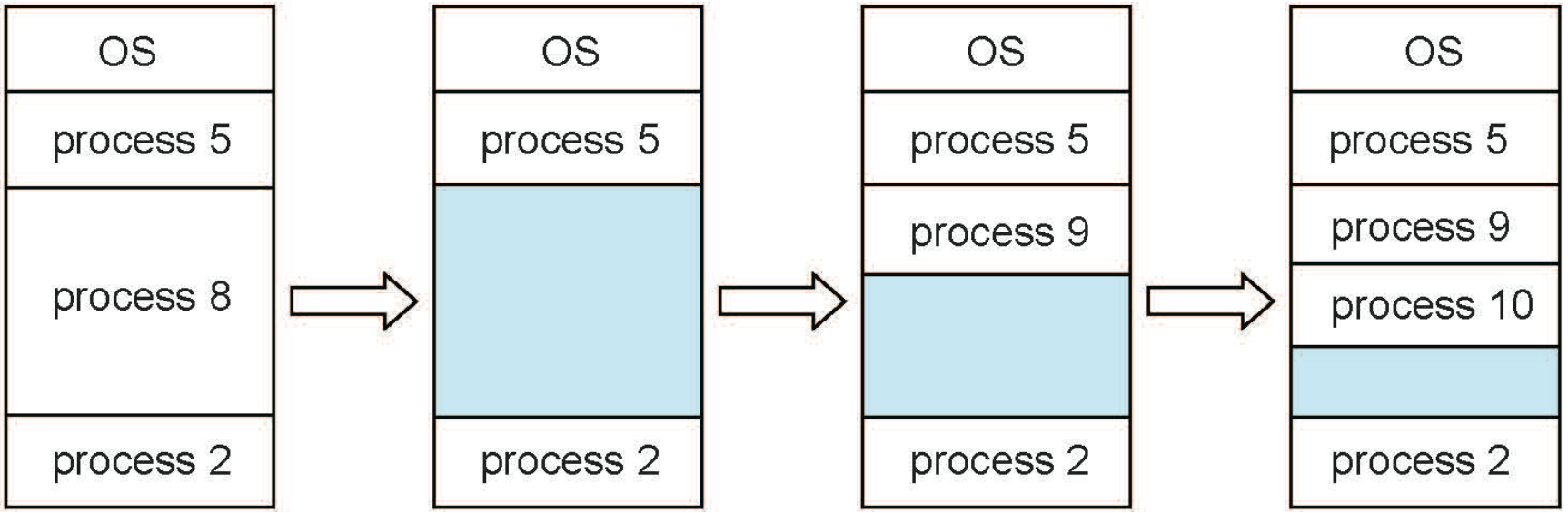
Multiple-Partition Allocation

- Fixed-Sized Partitions
  - Each partition only one process
  - Degree of multiprogramming limited by number of partitions
  - Originally used in IBM OS/360
  - No longer in use

Multiple-Partition Allocation (cont.)

- Variable Partition Scheme
  - 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
  - OS maintains information about:
    a) allocated partitions b) free partitions (hole)

Multiple-Partition Allocation (cont.)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **First-Fit**: Allocate ***first*** hole that is big enough
- **Best-Fit**: Allocate ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces smallest leftover hole
- **Worst-Fit**: Allocate ***largest*** hole; must also search entire list
  - Produces largest leftover hole

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

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**

Fragmentation (cont.)

- Solutions to External Fragmentation
  - **Compaction**
    - Shuffle memory contents to place all free memory together in one large block
      - E.g., move all used blocks to one end of memory
    - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - **Non-contiguous** memory allocation scheme
    - Segmentation
    - Paging
- Backing store same Fragmentation Problem

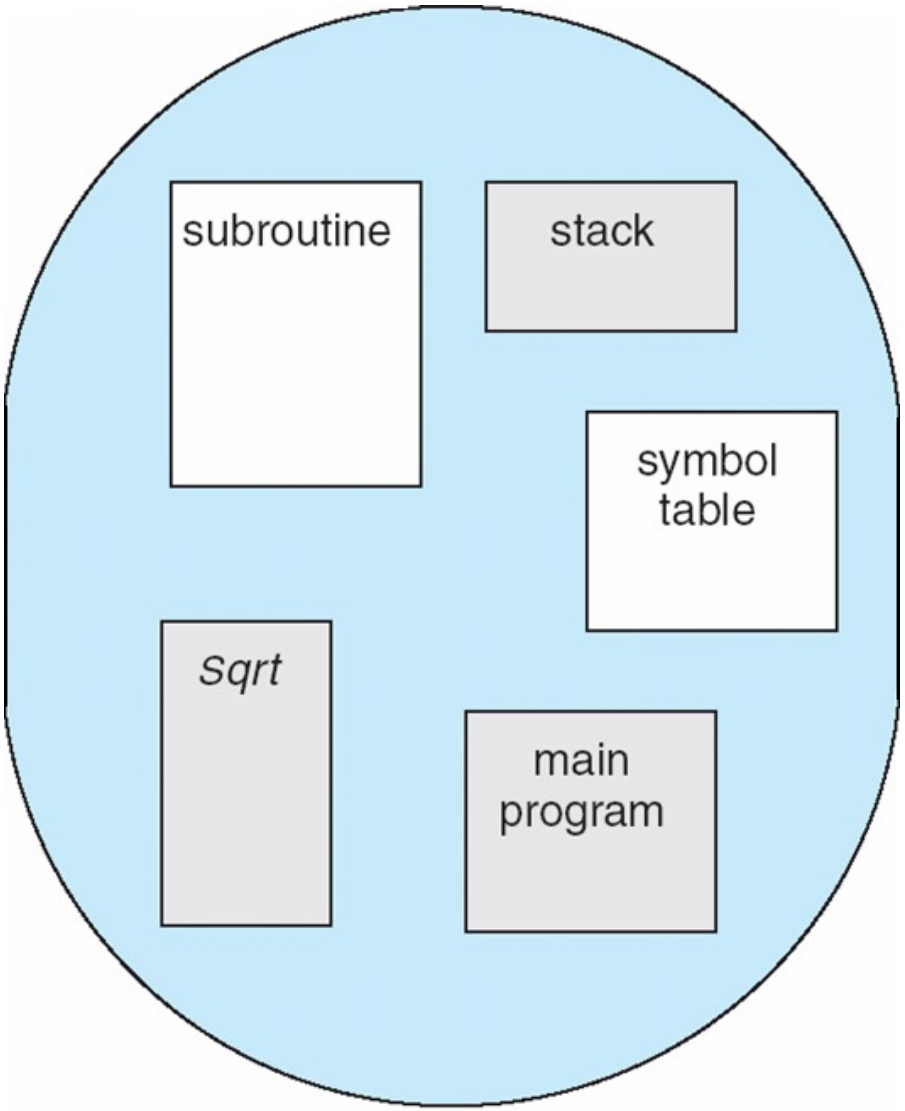
Segmentation

- Memory-Management Scheme that Supports User View of Memory
  - Maps programmer's view to physical memory
- A Program is a Collection of Segments
  - A Segment is a Logical Unit such as:

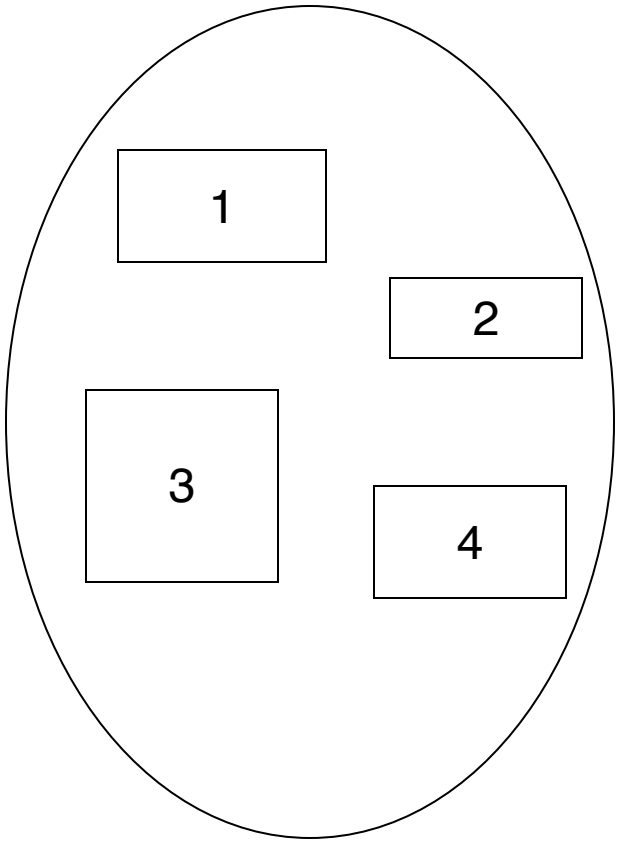
main program
procedure
function
method
object
local variables
global variables
common block
stack
symbol table
arrays

Code
Global variables
Heap
Stack
Standard C library

User's View of a Program



Logical View of Segmentation



User Space

Physical Memory Space

Segmentation Architecture

- Logical Address consists of a Two Tuple:

  <segment-number, offset>,

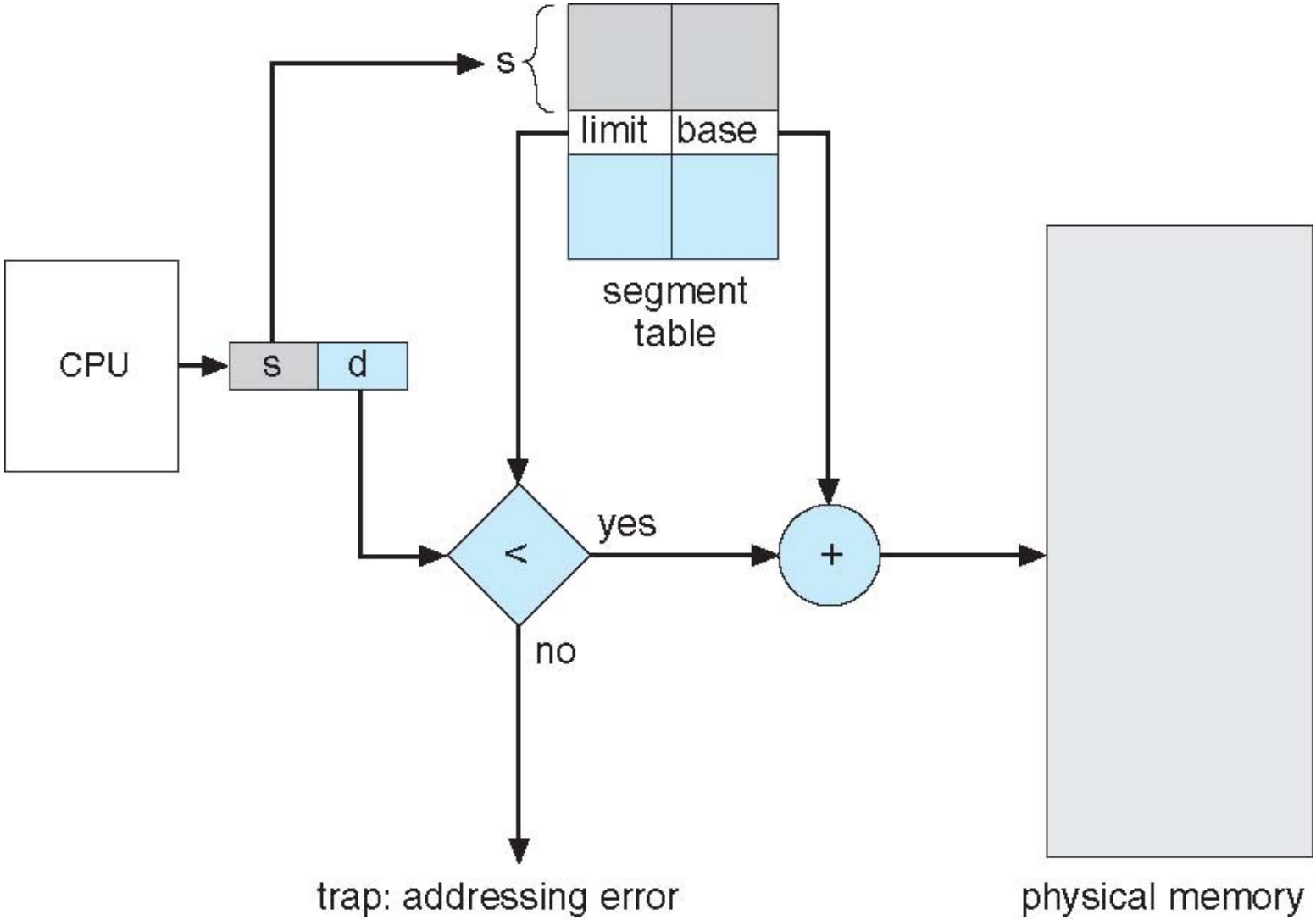
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains starting physical address where segments reside in memory
  - **limit** – specifies length of segment
- **Segment-table base register (STBR)** points to segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

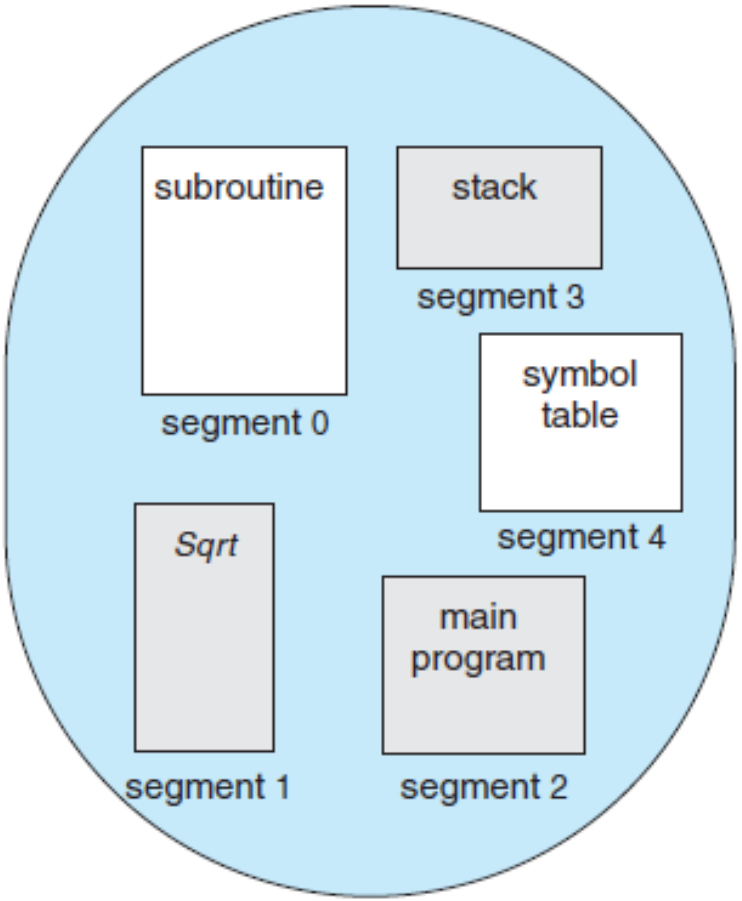
Segmentation Architecture (cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection Bits Associated with Segments
  - Code sharing occurs at segment level
- Segments Vary in Length
  - ➔ Memory Allocation is a Dynamic Storage-Allocation Problem

Segmentation Hardware

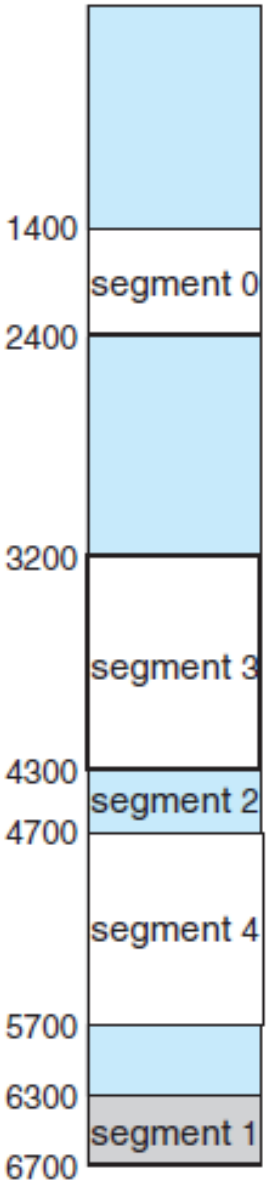


Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Segmentation vs. Variable-Sized Contiguous Allocation

- Variable-Sized Contiguous Allocation
  - Needs to bring entire process into memory
    - Both code, data, stack, …
- Segmentation
  - Break program into different segments
  - Brings segments to memory on demand
  - No need to bring unused library methods or unused code & data segments to memory

Problems with Segmentation?

- Example
  - Code segment: 10MB
  - Data segment: 100MB
  - Stack segment: 20MB
  - Lib segment: 10MB
- Problem?
  - Still significant amount of fragmentation
    - Both **external** & **internal** fragmentation

Paging

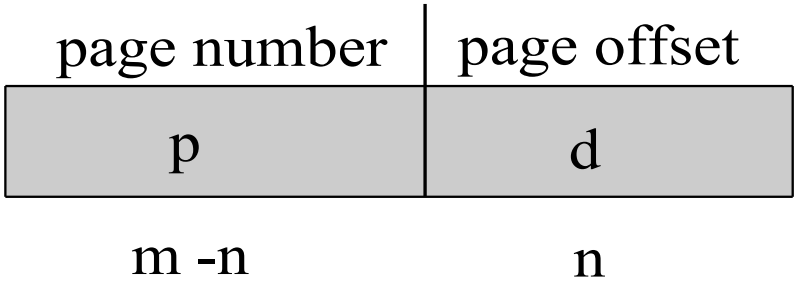
- Physical Address Space of a Process can be Non-Contiguous
  - Process is allocated physical memory whenever latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide Physical Memory into Fixed-sized blocks called **Frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes

Paging (cont.)

- 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

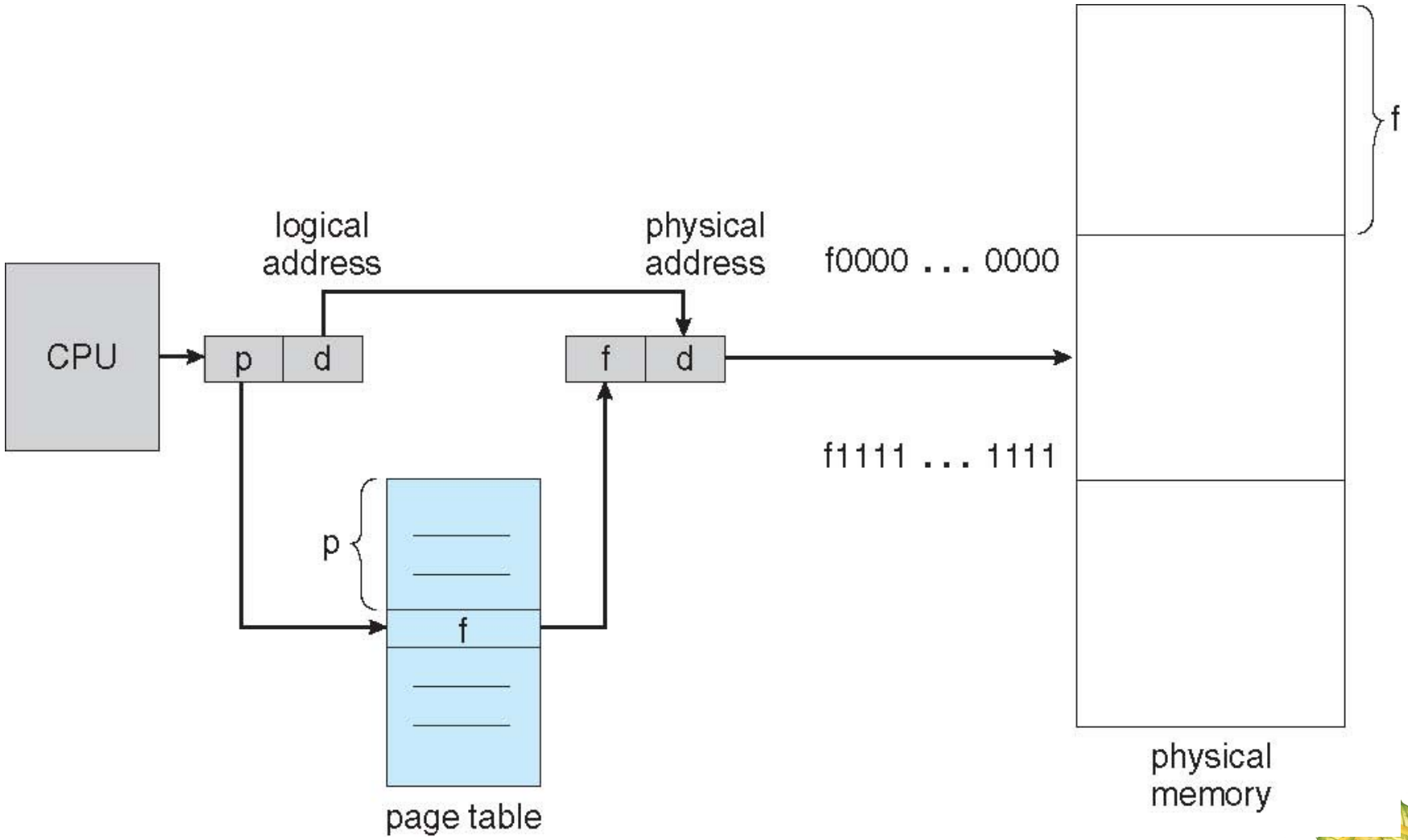
Address Translation Scheme

- Address Generated by CPU 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 physical memory address that is sent to memory unit

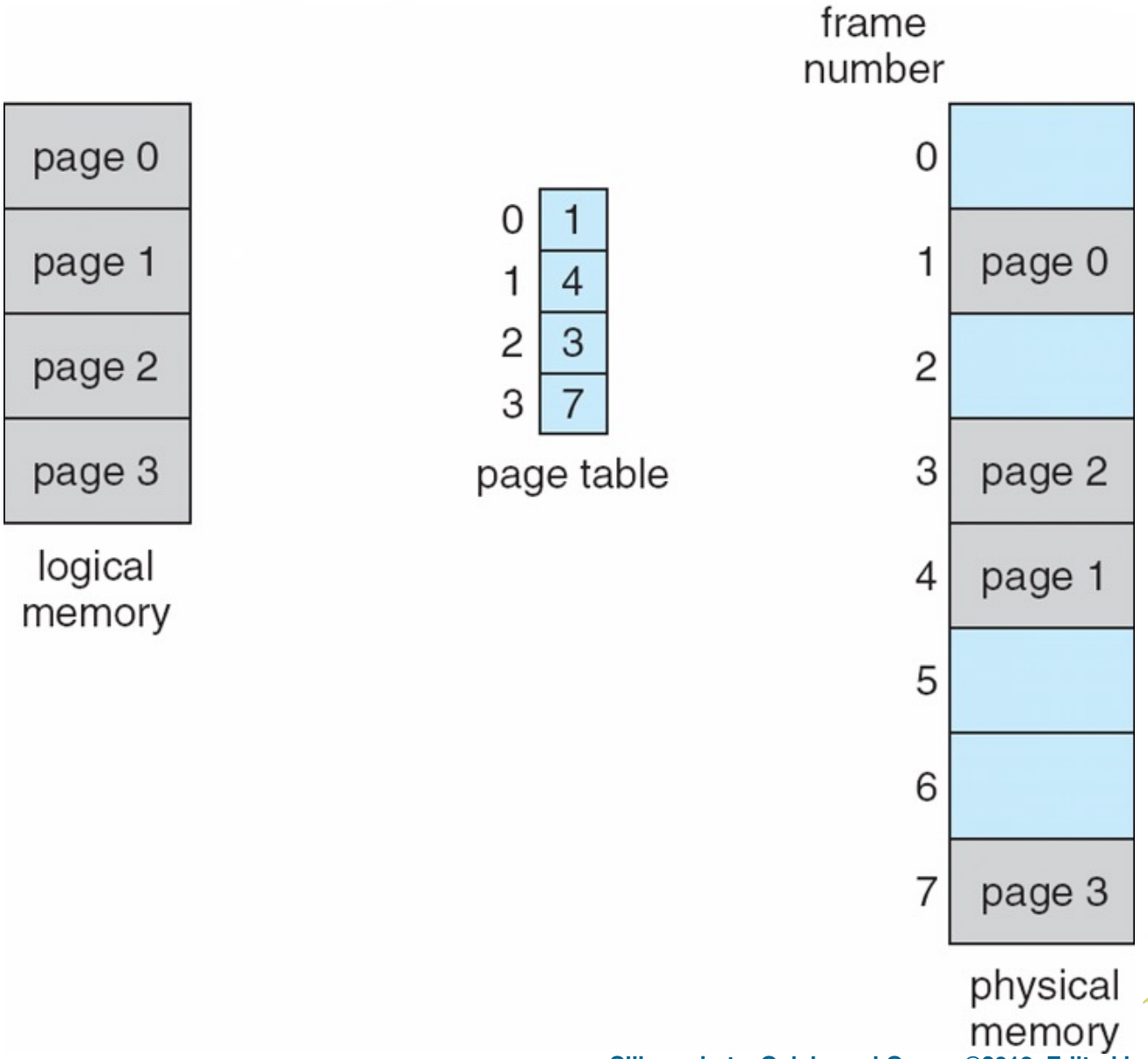


    - Logical address space: 2^m
    - Page size: 2^n

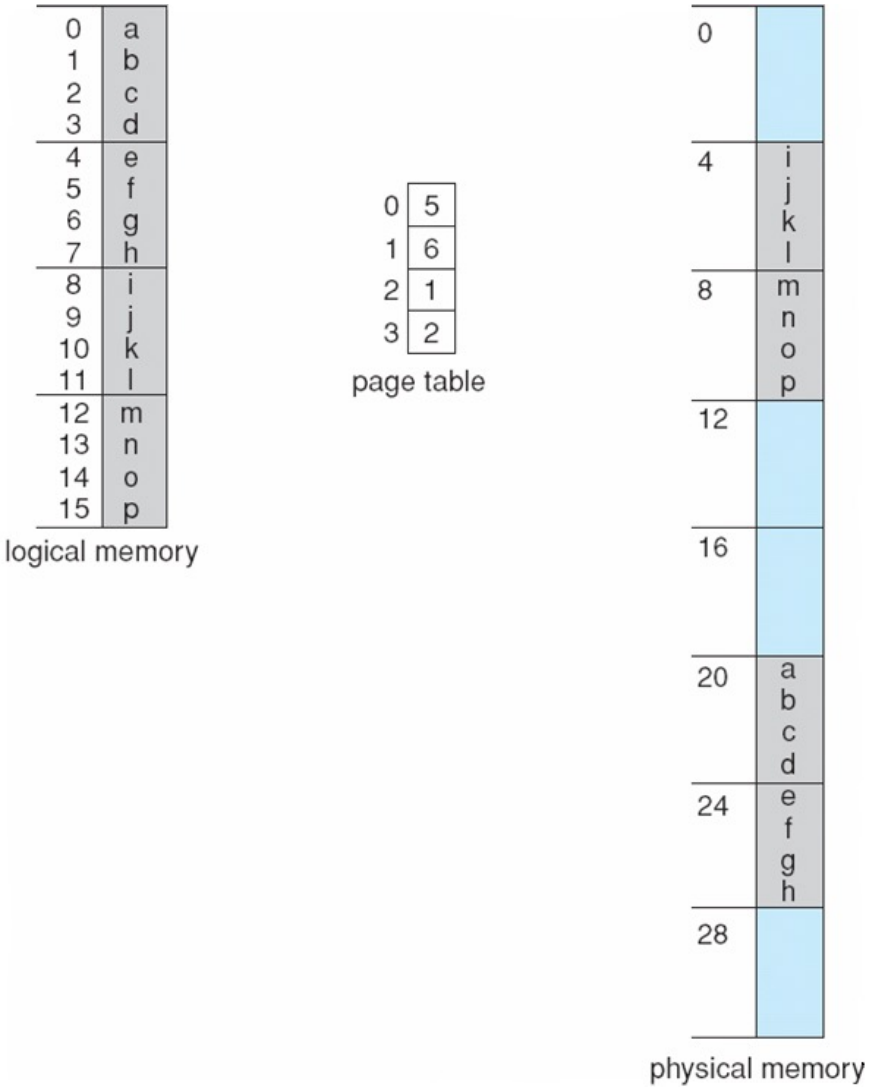
Paging Hardware



Paging Model of Logical and Physical Memory



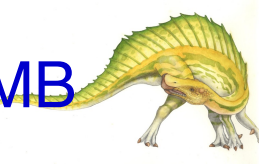
Paging Example



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

Paging (cont.)

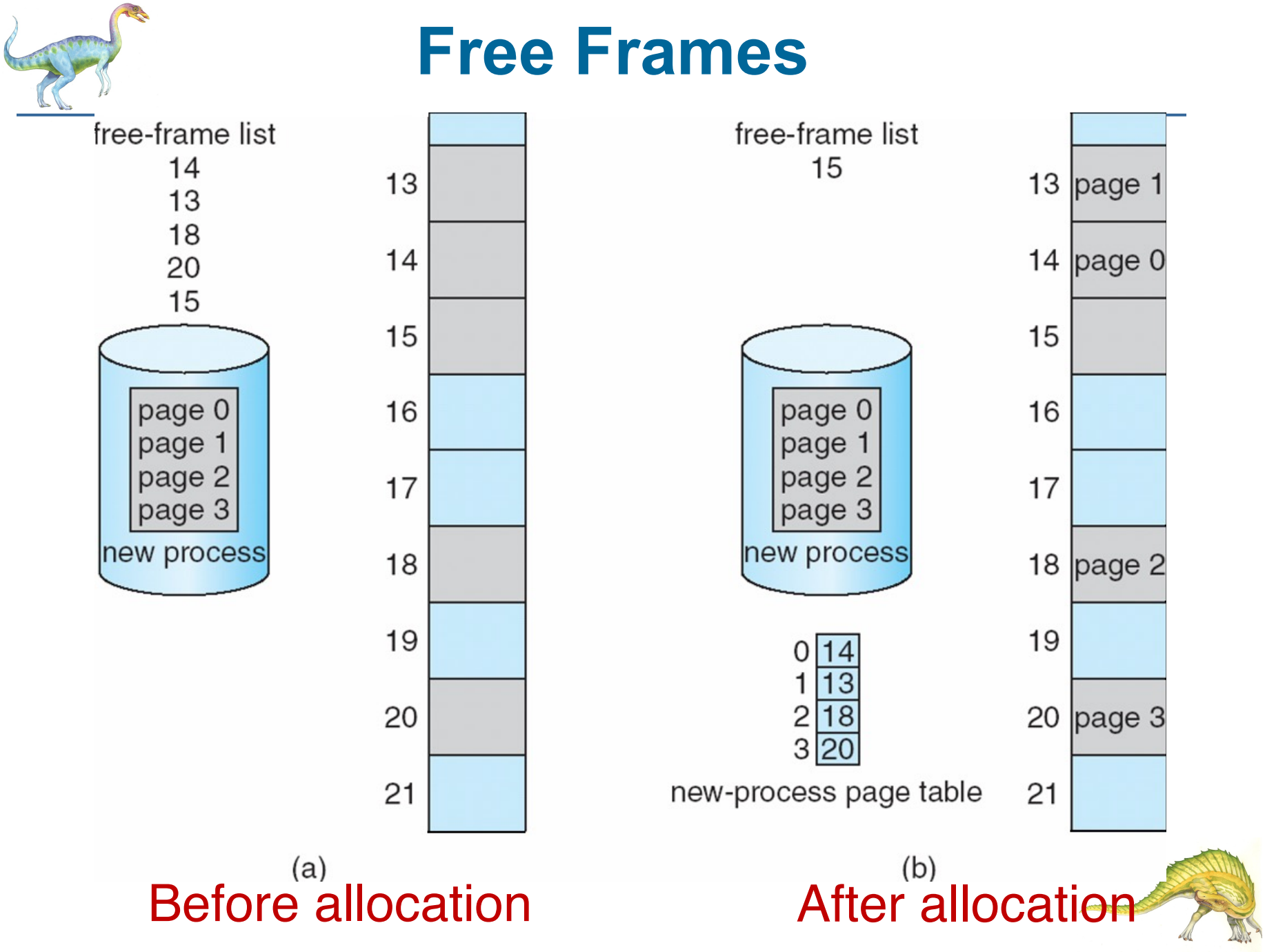
- Calculating Internal Fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048-1,086 = 962B
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB



Paging (cont.)

- Process View and Physical Memory now Very Different
- By Implementation, Process can only Access its Own Memory

Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
Before allocation
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)
After allocation

Implementation of Page Table

- Page Table Kept in Main Memory
- **Page-Table Base Register** (**PTBR**) Points to Page Table
- **Page-Table Length Register** (**PTLR**) Indicates Size of Page Table
- In this Scheme every Data/Instruction Access Requires **two Memory Accesses**
  - One for page table and one for data/instruction
- Two Memory Access Problem can be Solved by Use of a Special Fast-Lookup HW cache
  - Called **Associative Memory** or **Translation Look-Aside Buffers** (**TLBs**)

Implementation of Page Table (cont.)

- Fully Associative TLB
  - Fast ☺
  - Very costly ☹
  - Typically 64~1024 entries
- TLBs work similar to Caches
  - Consists of two parts: Key (Tag) and a Value
- TLB Functionality
  - An incoming item compared with all keys simul.
  - If found, corresponding value filed is returned
  - If not found ➔ TLB miss

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
- On a TLB Miss, Value Loaded into TLB for Faster Access Next Time
  - Replacement policies must be considered
  - Some entries **wired down** for permanent fast access (e.g., kernel code pages)

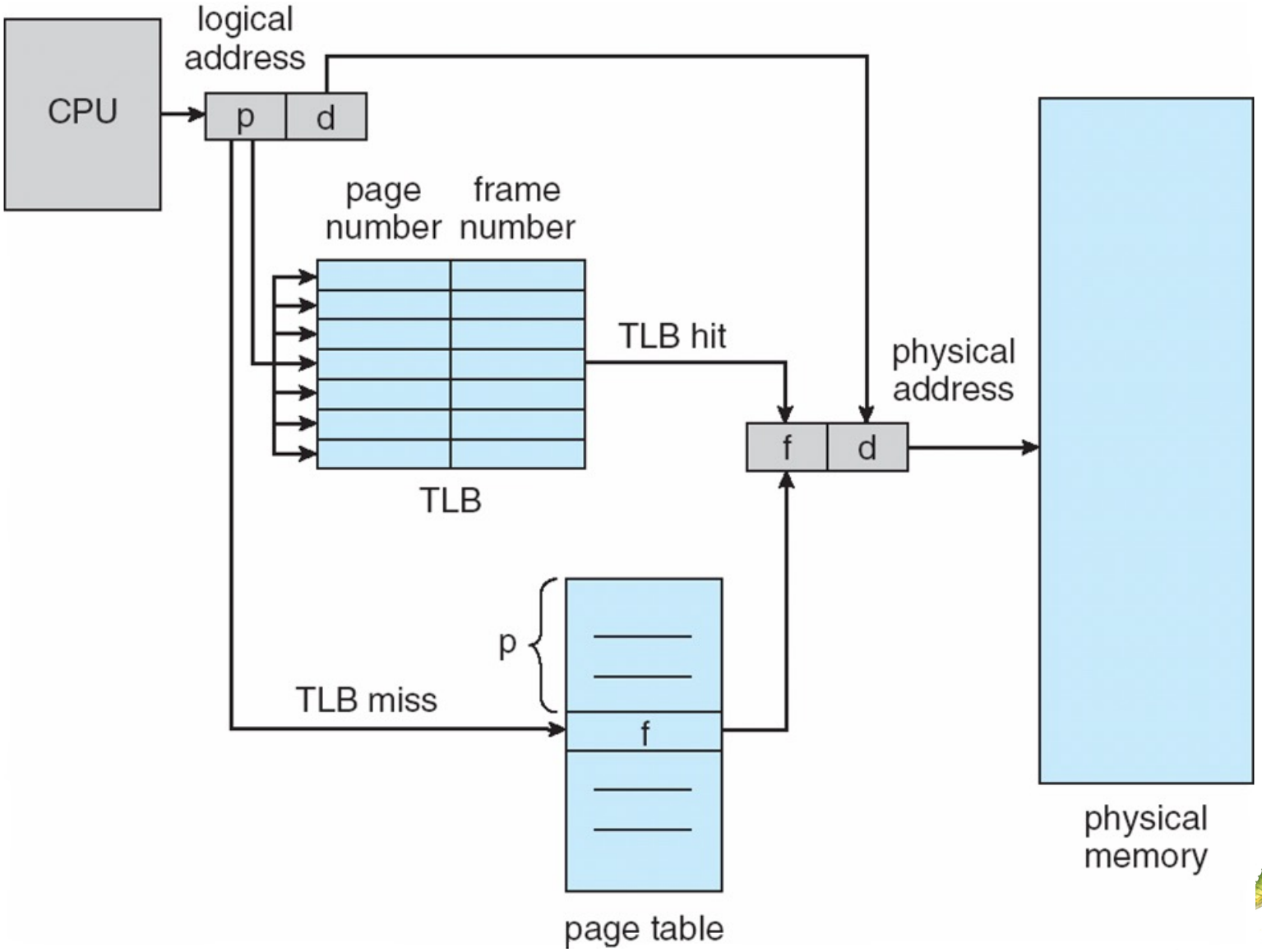
Associative Memory

- 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



Effective Access Time

- Associative Lookup time unit (ε)
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

Effective Access Time (cont.)

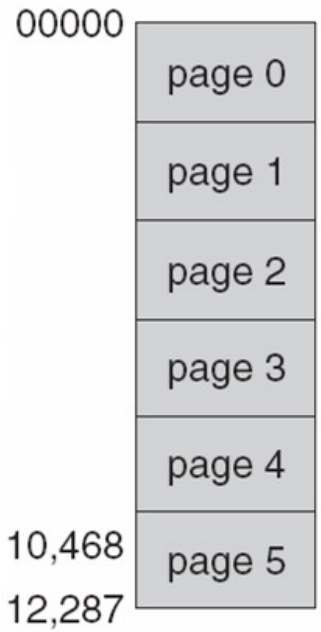
- Example 1:
  - Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Example 2:
  - Consider more realistic hit ratio
  - α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

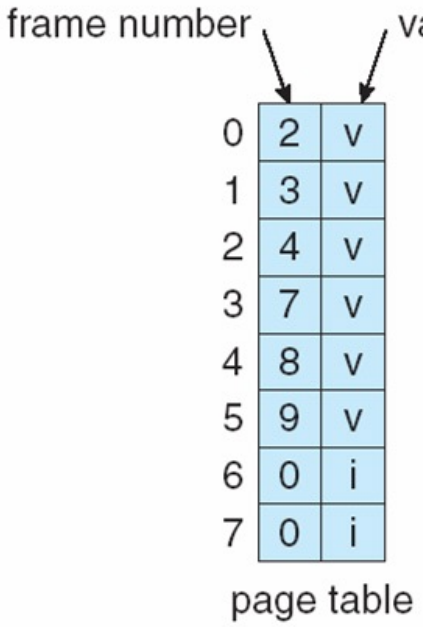
Memory Protection

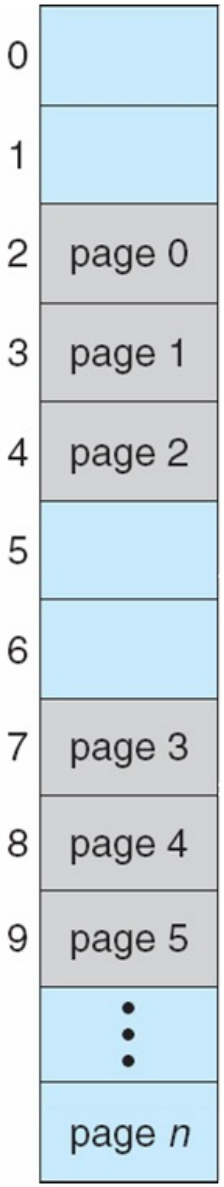
- Memory Protection Implemented by Associating Protection bit with each Frame to Indicate if Read-Only or Read-Write Access is Allowed
  - Can also add more bits to indicate page execute-only
- **Valid-Invalid** bit Attached to each Entry in Page Table
  - "valid" indicates that associated page is in process' logical address space, and is thus a legal page
  - "invalid" indicates that page is not in process' logical address space
  - **Or** use **page-table length register** (**PTLR**)
- Any Violations Result in a Trap to Kernel

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

Example: a process with logical addresses ranging from 0 to 10468





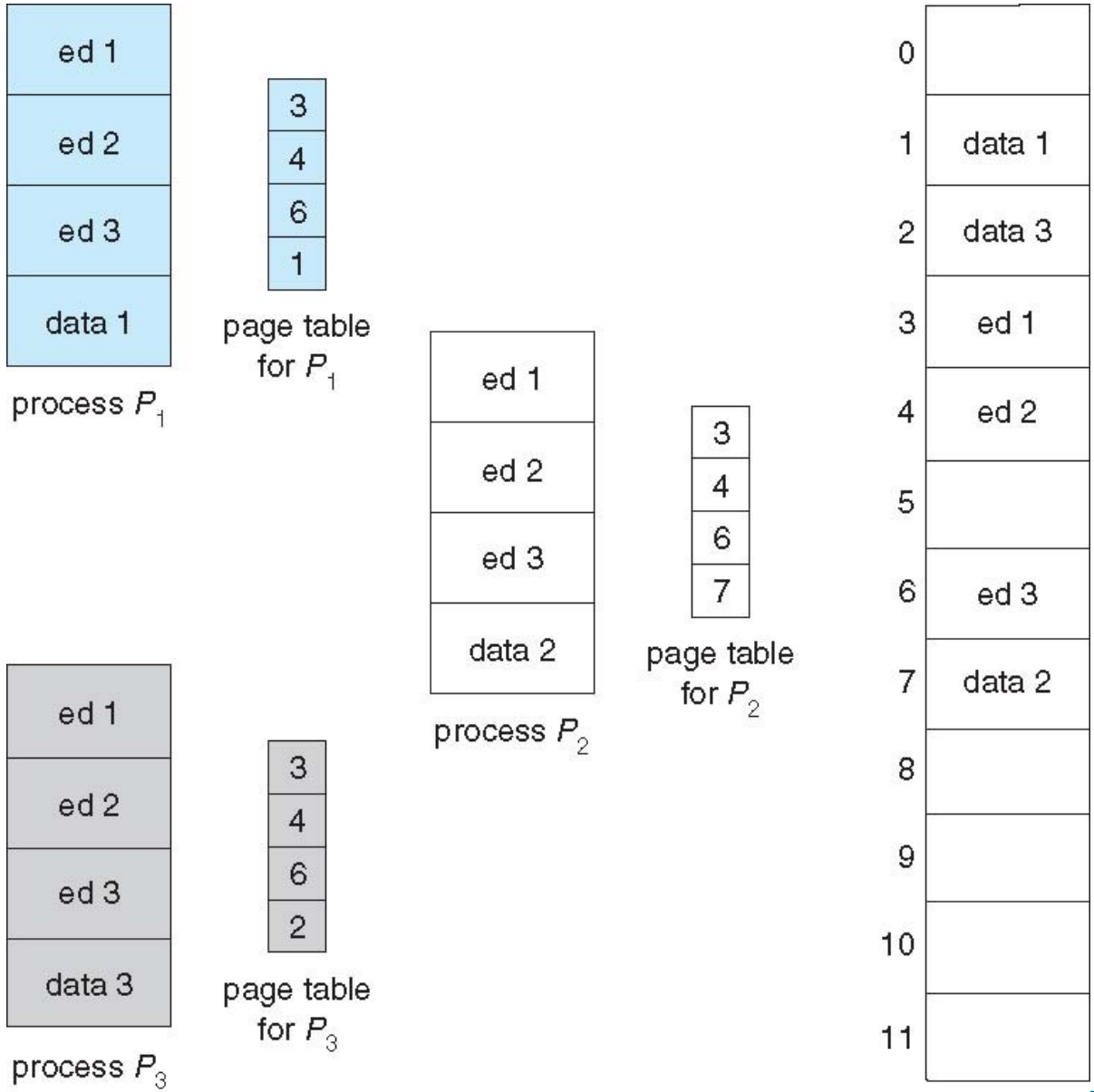


Any problem with this scheme?

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 same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private Code and Data**
  - Each process keeps a separate copy of code and data
  - Pages for private code and data can appear anywhere in logical address space

Shared Pages Example



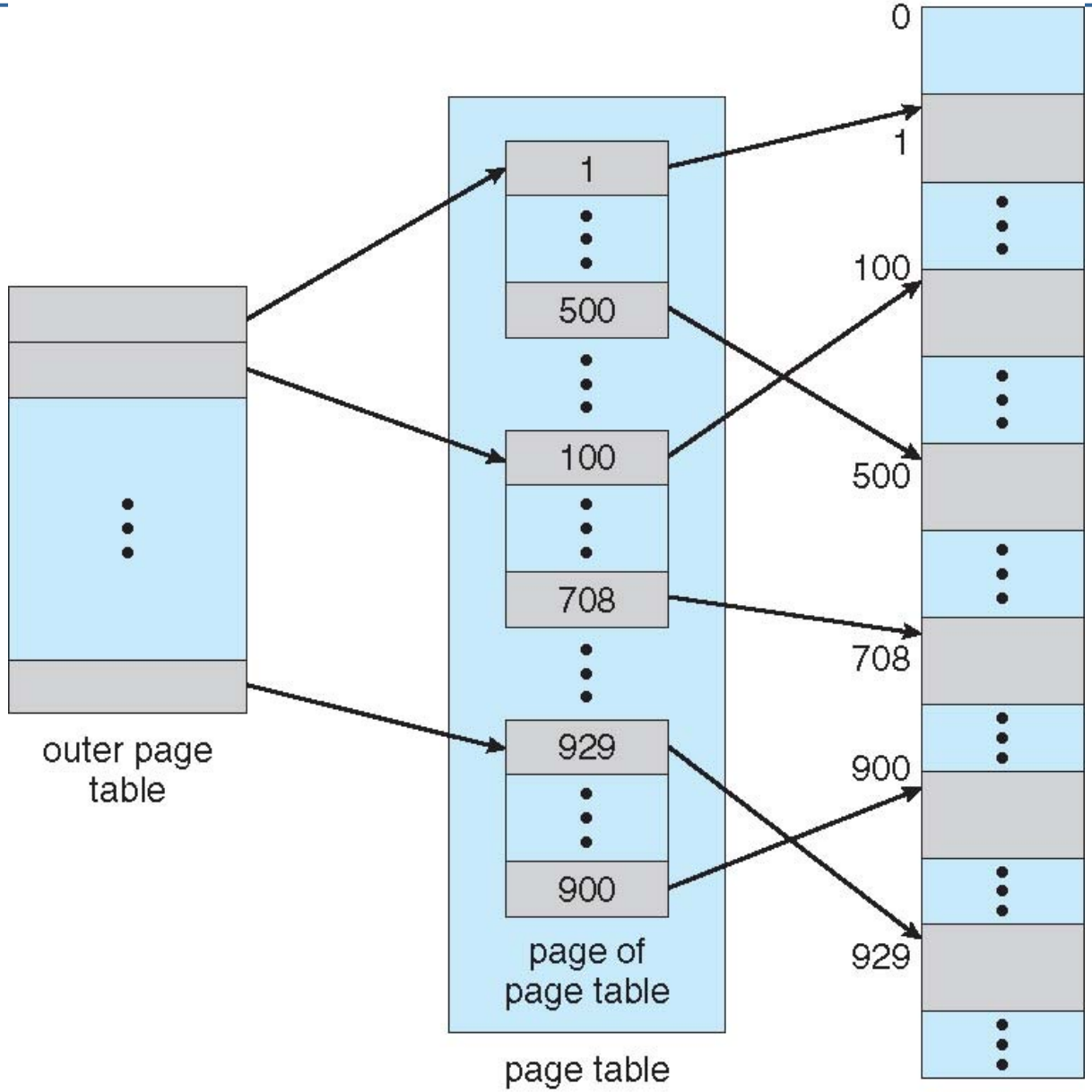
Structure of Page Table

- 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 Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

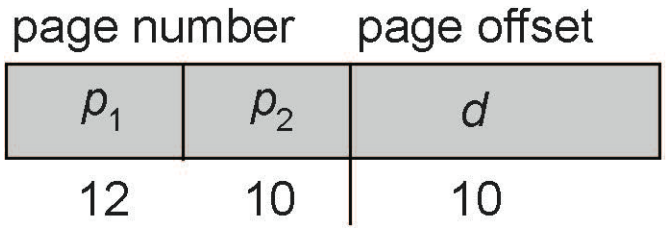
- Break up Logical Address Space into Multiple Page Tables
- A Simple Technique is a Two-Level Page table
- We then Page the page table

Two-Level Page-Table Scheme



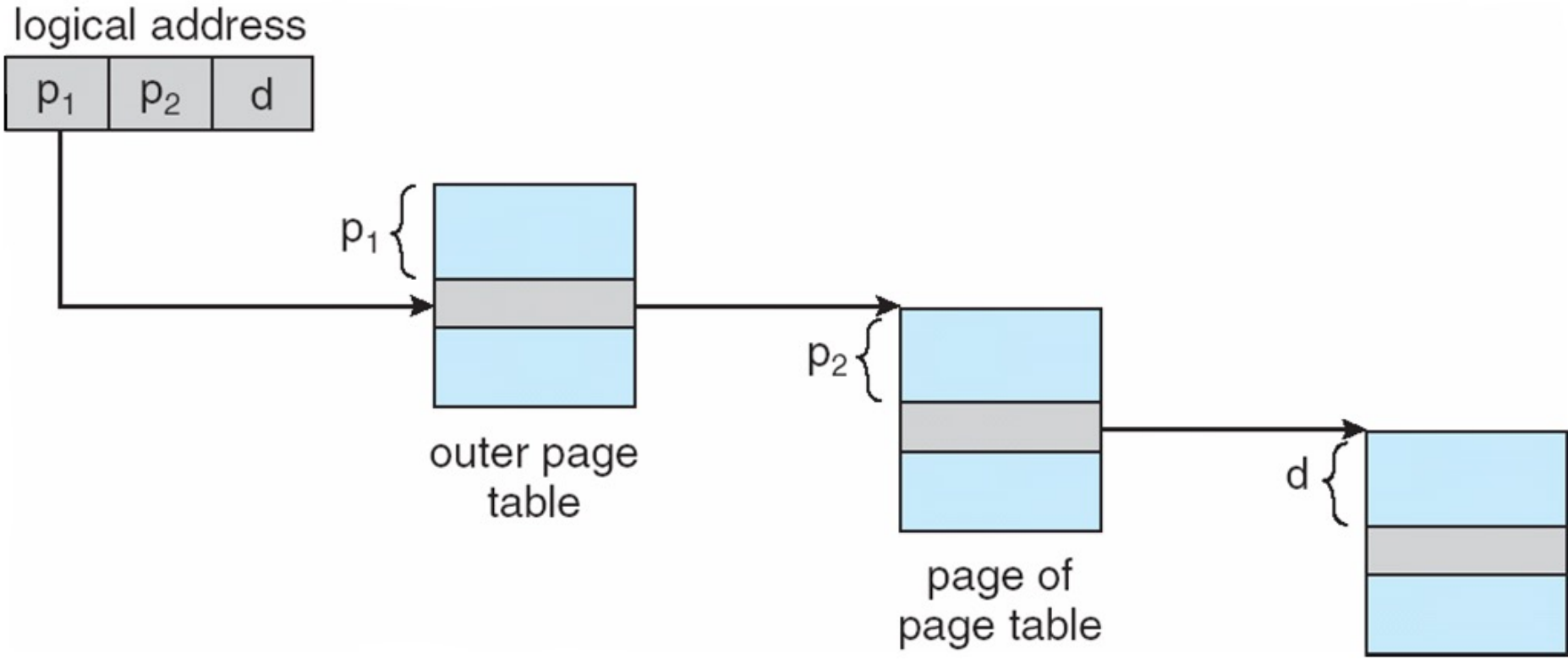
Two-Level Paging Example

- A Logical Address (on 32-bit machine with 1K page size) Divided into:
  - A page number consisting of 22 bits
  - A page offset consisting of 10 bits
- Since Page Table is paged, page Number is further Divided into:
  - A 12-bit page number
  - A 10-bit page offset



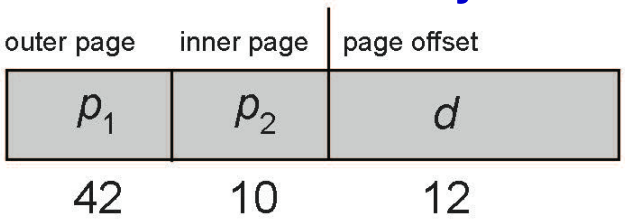
- Thus, a Logical Address is as follows:
  - p_1 is an index into outer page table
  - p_2 is displacement within page of inner page table
    - Known as **forward-mapped page table**

Address-Translation Scheme



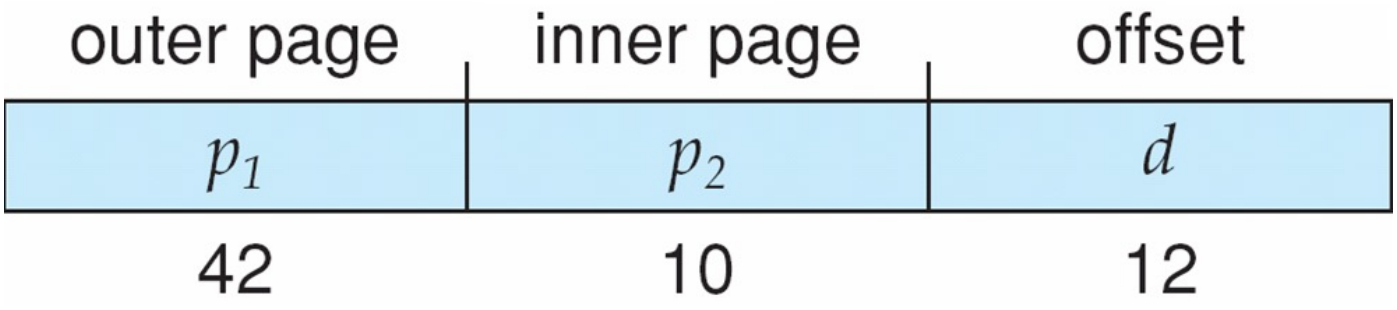
64-bit Logical Address Space

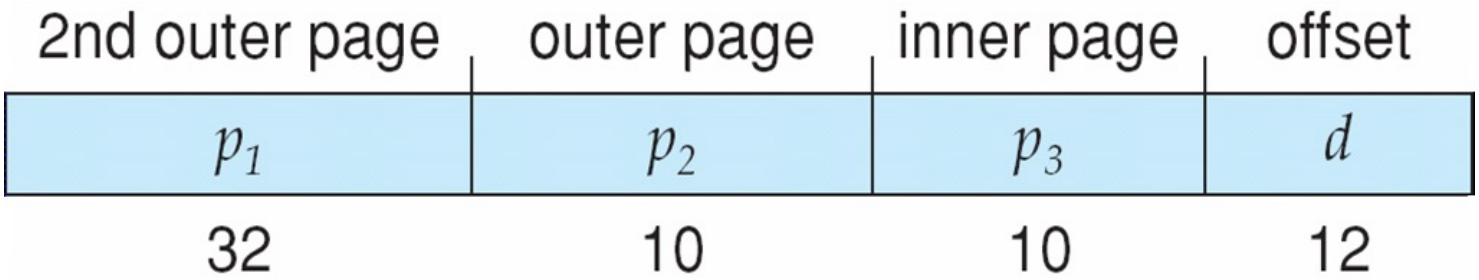
- Even two-level paging scheme not Sufficient
- If Page Size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If 2-level scheme, inner page tables have 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - Possibly 4 memory access to get to one physical memory location ➔ hierarchical paging not appropriate for 64-bit addressing

Three-Level Paging Scheme

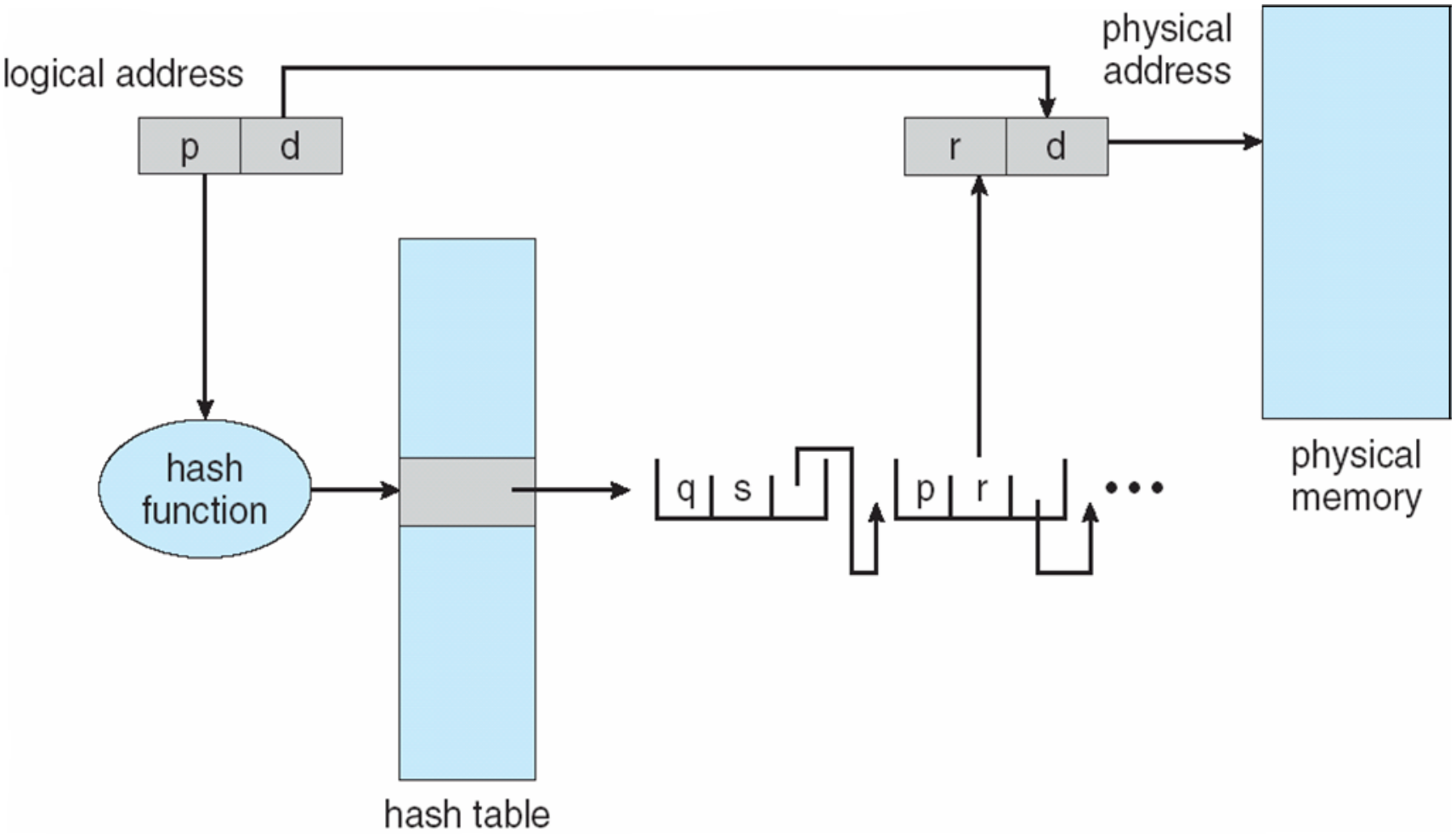




Hashed Page Tables

- Common in Address Spaces greater than 32 bits
- Virtual Page Number Hashed into a Page Table
  - This page table contains a chain of elements hashing to same location
- Each element contains (1) virtual page number (2) value of mapped page frame (3) a pointer to next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, corresponding physical frame is extracted

Hashed Page Table



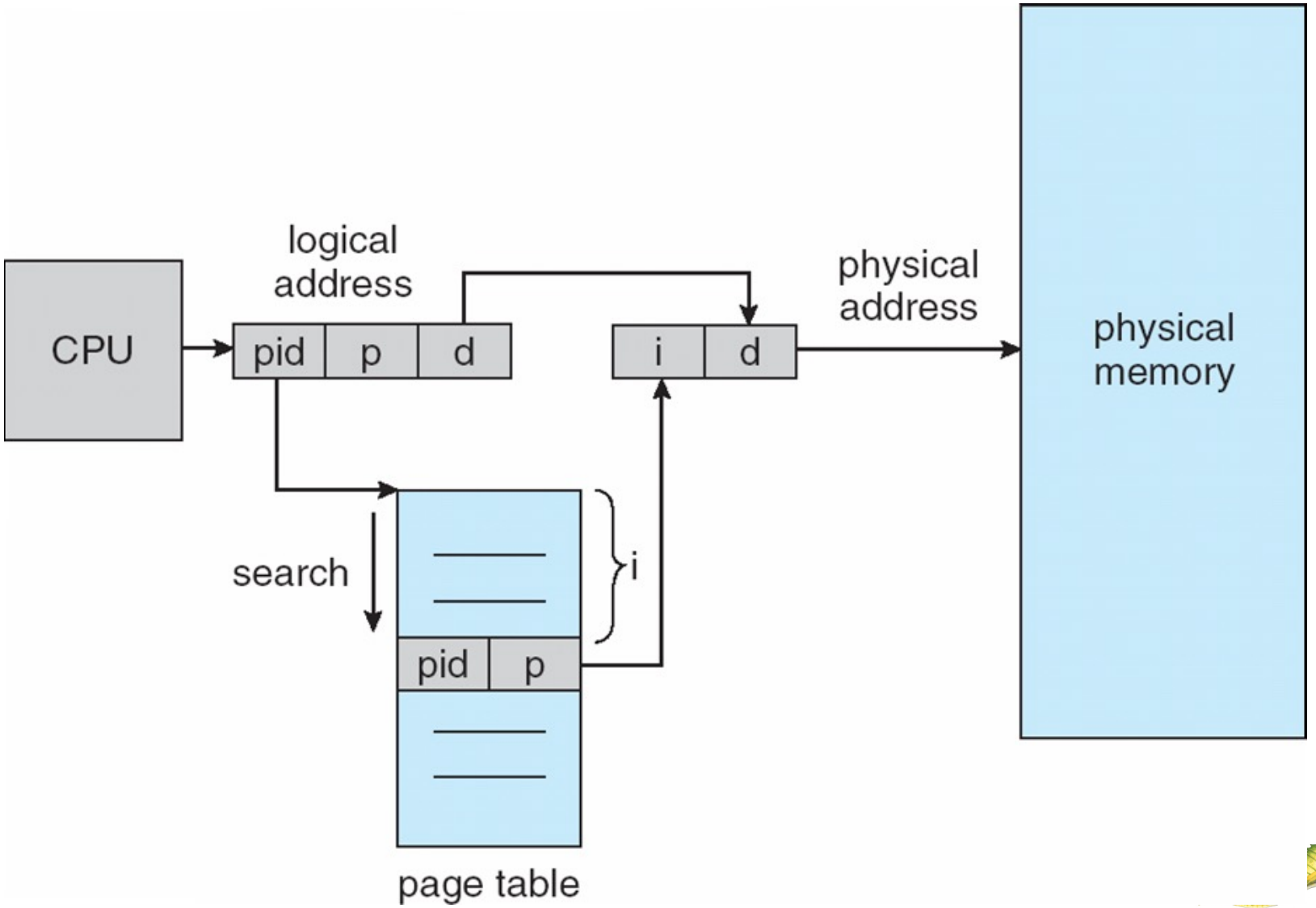
Inverted Page Table

- One Entry for each Real Page of Memory
  - Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- Entry consists of virtual address of page stored in that real memory location, with information about process that owns that page
- Pros
  - Decreases memory needed to store each page table

Inverted Page Table (cont.)

- Cons
  - Increases time needed to search table when a page reference occurs
    - Stored by physical addresses
- Use Hash Table to Limit Search to one — or at most a few — page-table entries
  - TLB can accelerate access

Inverted Page Table Architecture



Example: Intel 32 and 64-bit Architectures

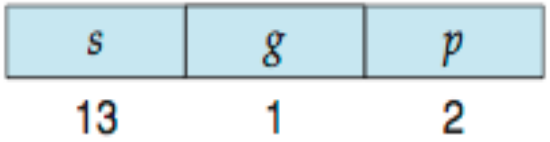
- Dominant Industry Chips
- Pentium CPUs are 32-bit
  - Called IA-32 architecture
- Current Intel CPUs are 64-bit
  - Called IA-64 architecture

Example: Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4GB
  - Up to 16K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (LDT))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (GDT))

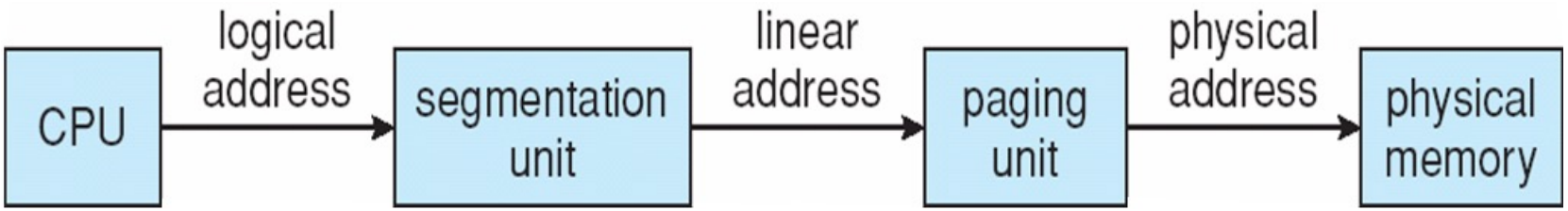
Example: Intel IA-32 Architecture (cont.)

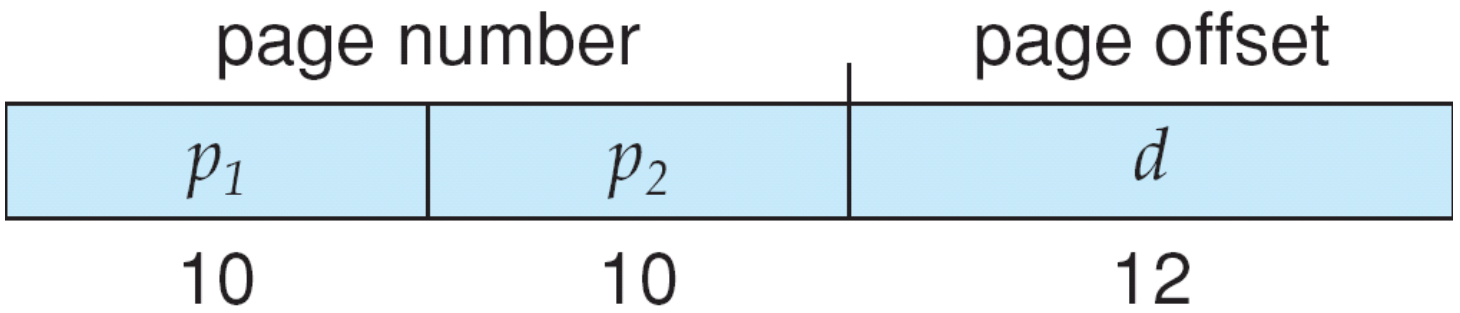
- CPU Generates Logical Address
  - Selector given to segmentation unit
    - Which produces linear addresses



  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4KB or 4MB

Logical to Physical Address Translation in IA-32





Reading Assignment

- Clustered Page Table
- Paging in Oracle SPARC Solaris
- Detailed Paging in IA-32 and IA-64

Hashed Page Tables (cont.)

- 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)

Oracle SPARC Solaris

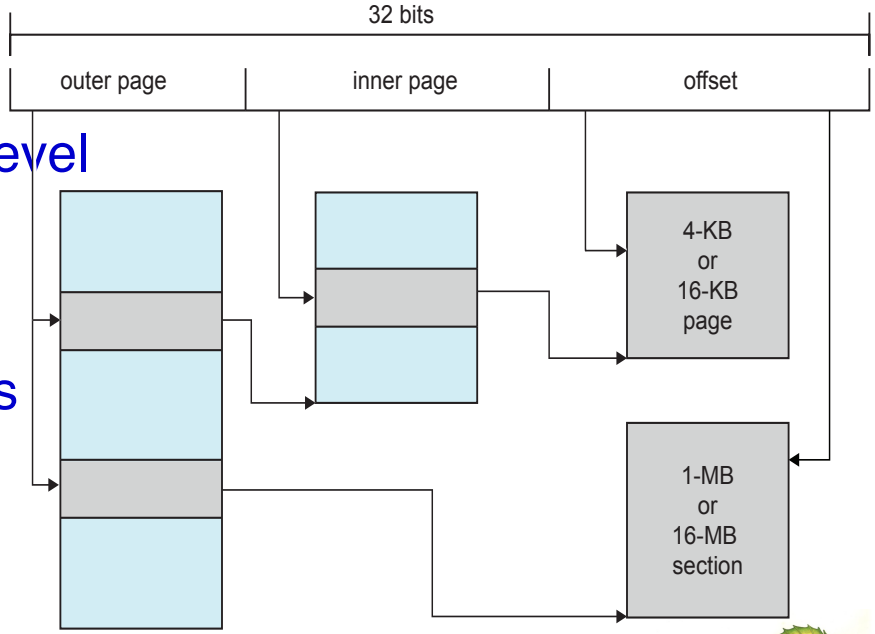
- Consider modern, 64-bit OS example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two Hash Tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

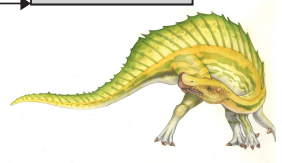
Oracle SPARC Solaris (cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

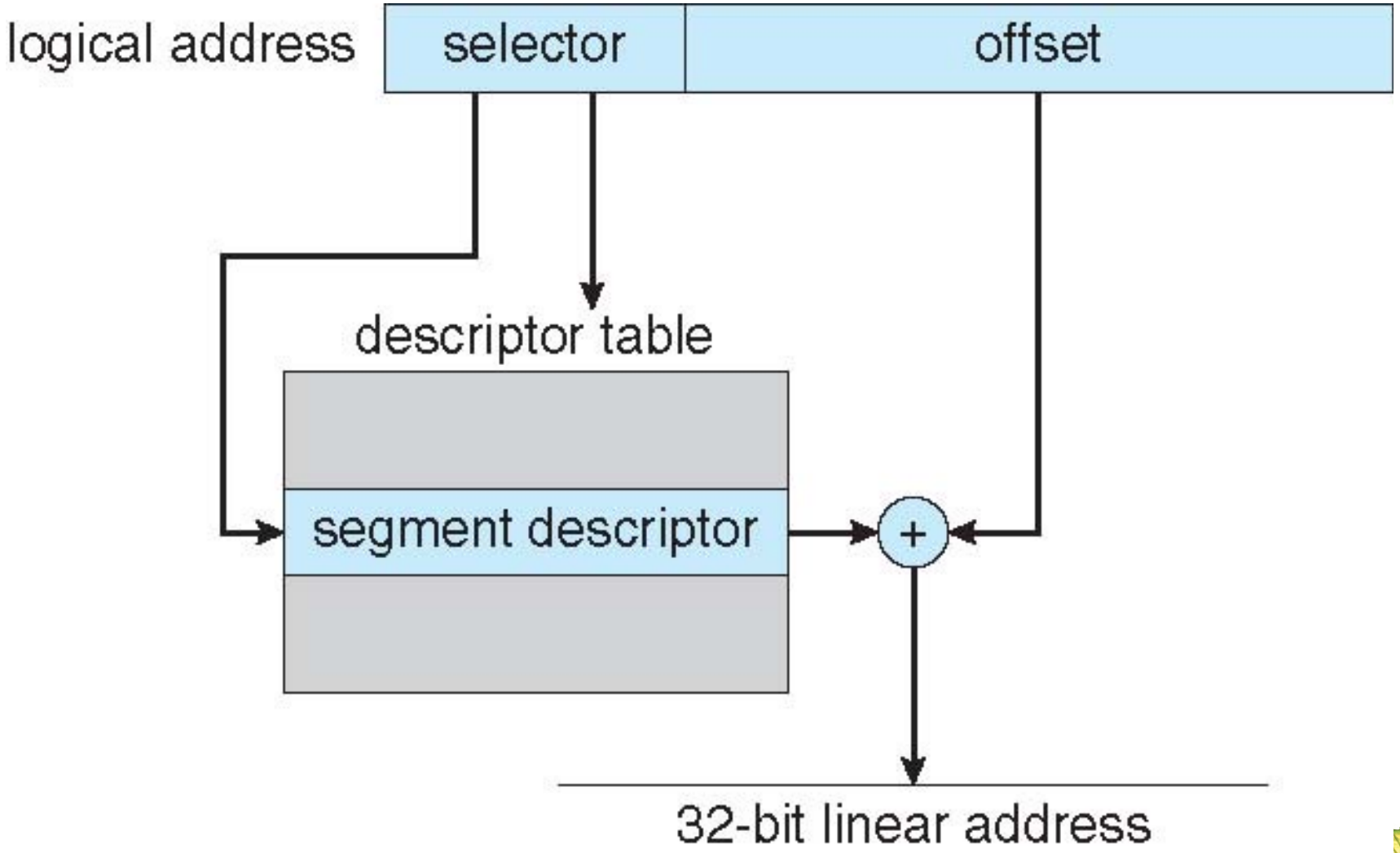
Example: ARM Architecture

- 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

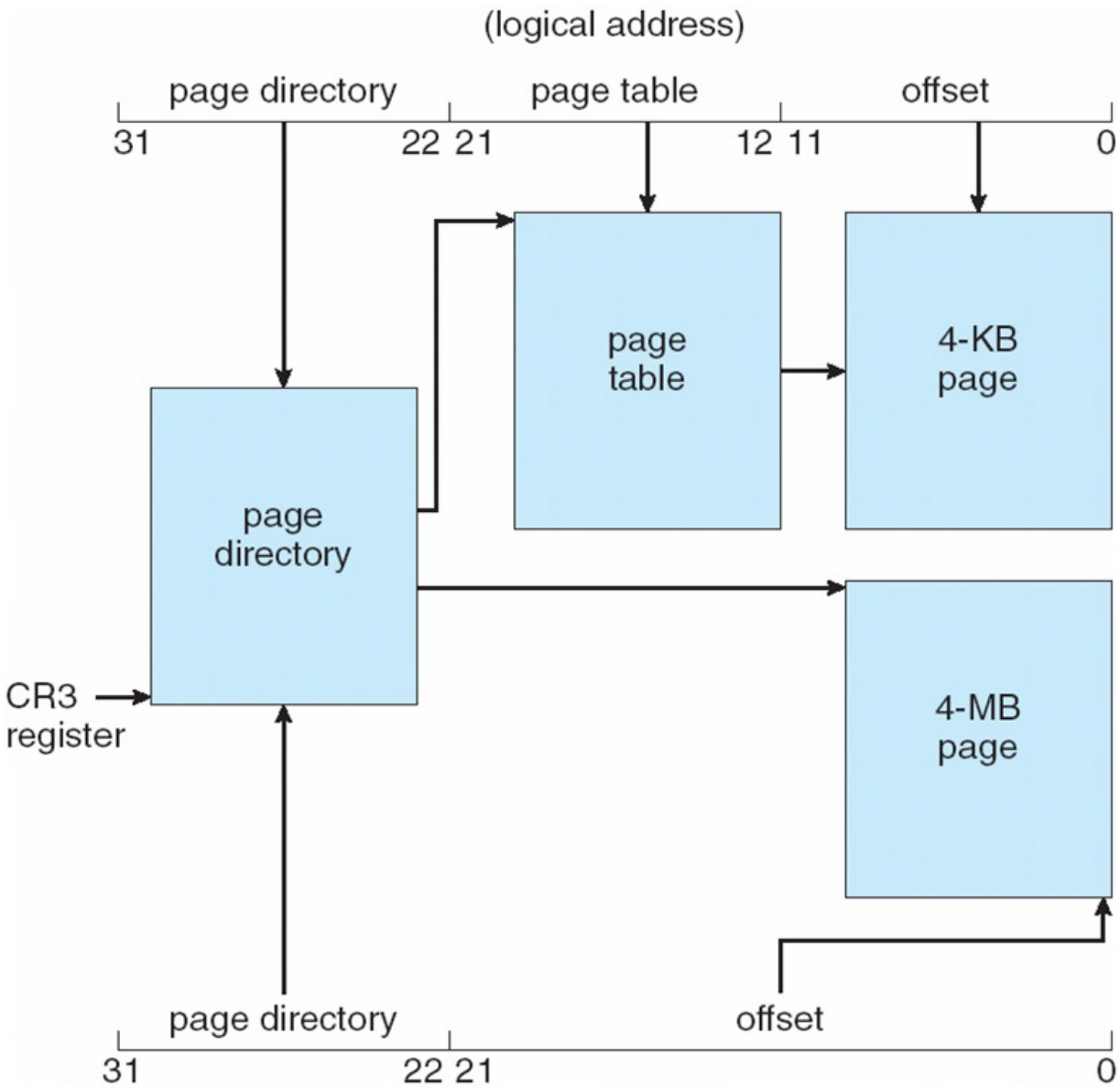




Intel IA-32 Segmentation

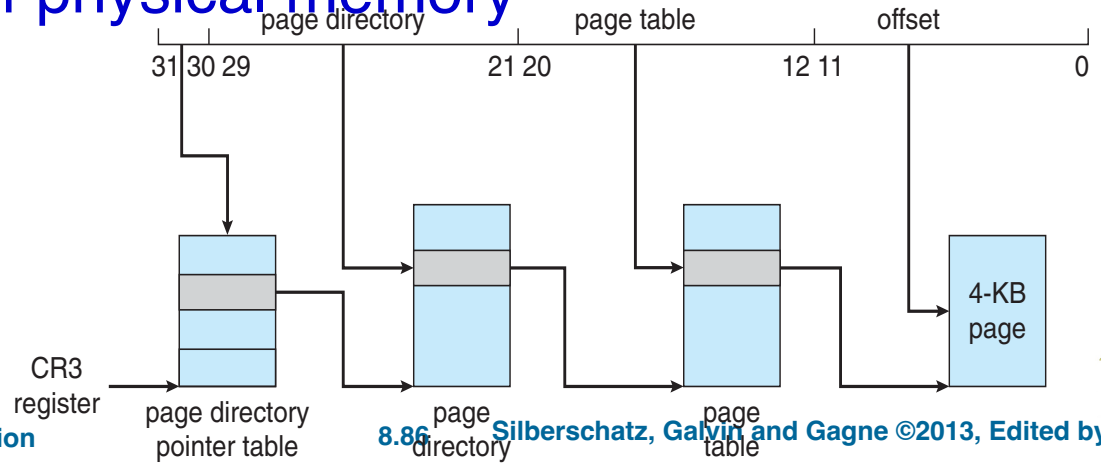


Intel IA-32 Paging Architecture



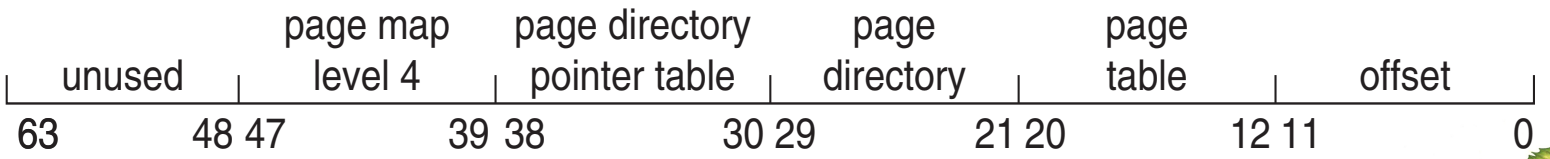
Intel IA-32 Page Address Extensions

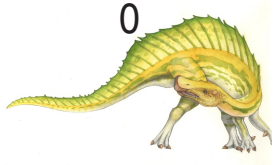
- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



Intel x86-64

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





End of Lecture 8