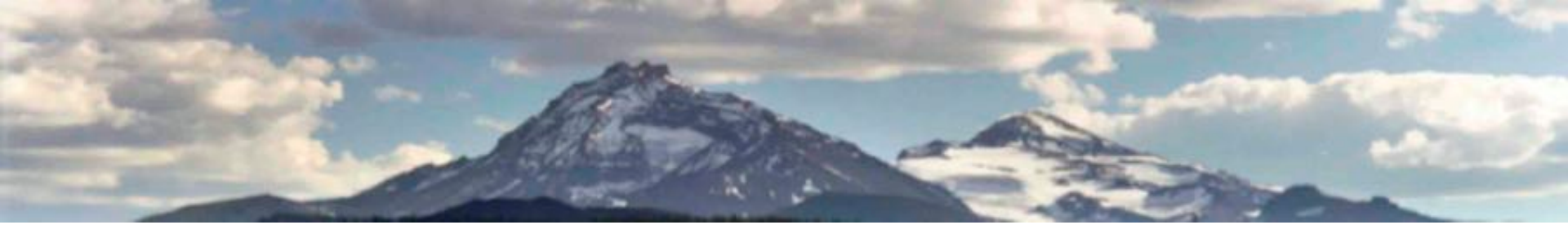


# CIS 415
# Operating Systems
# Main Memory

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2020

UNIVERSITY OF OREGON

# *Logistics*

- Congratulations on the midterm!
  - Hope everything went ok in taking the test
  - Hope you thought it was a fair evaluation
  - Questions were shuffled in each section
  - I will try to have them graded by next Tuesday
- Keep working hard on Project 2
  - Lab sessions will focus on the project

# *Outline*

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Examples
  - The Intel 32 and 64-bit Architectures
  - 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 give a detailed example pure segmentation and segmentation with paging

# *Background*

- Program must be brought (from external storage) into memory and "loaded" in a process for it to be run
  - A program goes through several steps before producing an executable code image that the OS uses to create a process
- A program starts with "logical" addresses that are in the "logical" address space its process
  - Instructions produce addresses with respect to logical addresses (also called "virtual" addresses)
  - Need to bind (map between) "logical" addresses and "physical" addresses in memory
- All this depends on both software and hardware
  - Software includes compilers, loaders, and OS
  - Hardware include memory system, address translation, …

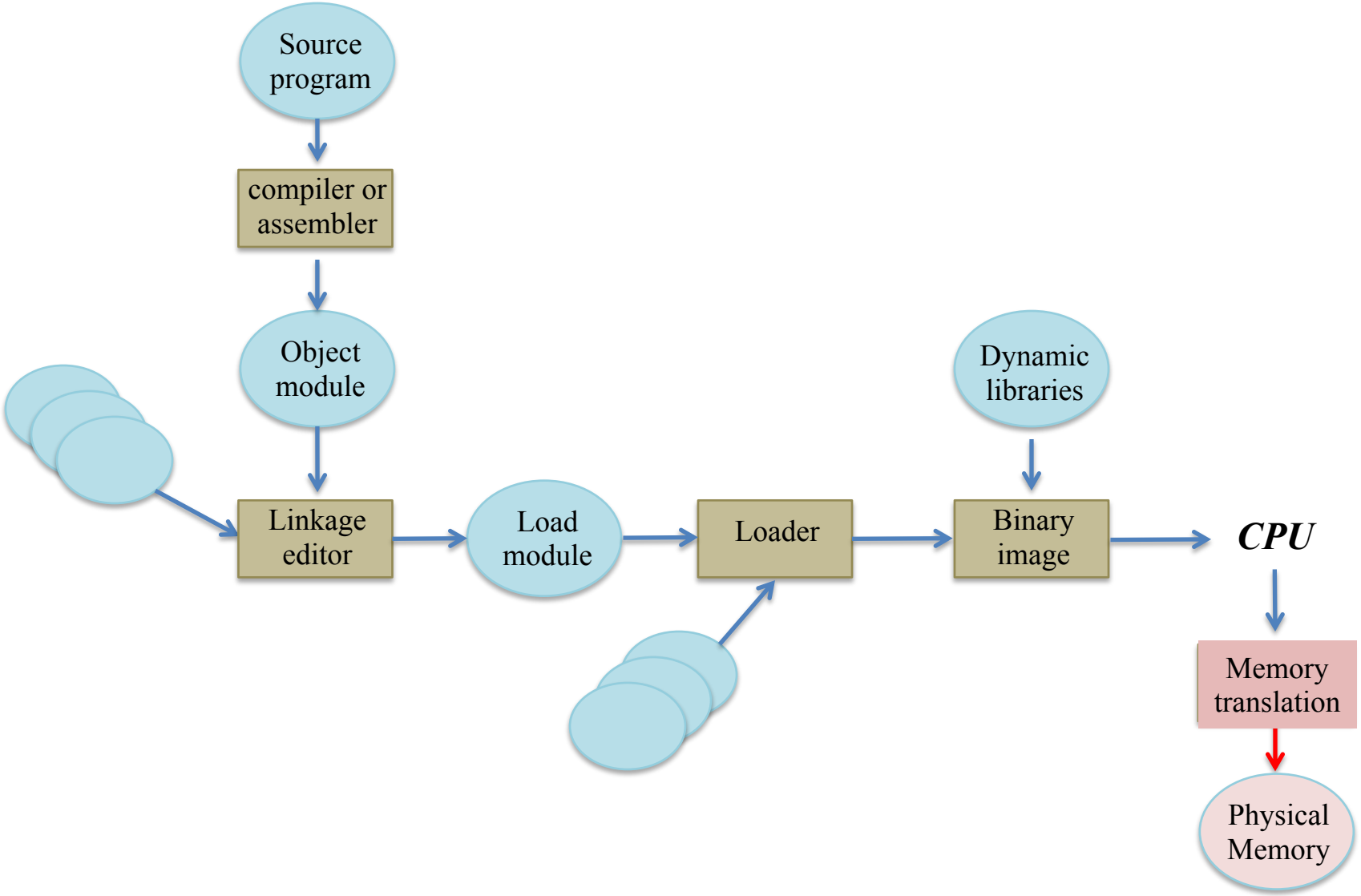
# *Address Binding*

- Addresses represented in different ways at different stages of a program's life
  - Need to bind one address to another
  - Source code addresses usually *symbolic* (i.e., names)
  - Compiled code has addresses that need to bind to *relocatable addresses*
  - Linker or loader will bind relocatable addresses to *absolute addresses*
  - Each binding maps one address space to another
- All of these addresses are "logical" adddresses

# *Memory and Address Binding*

- Main memory and CPU registers are the only storage directly addressable by a program
  - CPU registers are accessed in one CPU clock (or less)
  - Memory system sees addresses from read/write operations
- Addresses used in CPU instructions are logical addresses
  - Reference the logical address space of a process
  - These logical addresses must be "translated" to physical addresses
- What does address translation (address binding)?
  - *Compile time*: If a memory location is known *a priori* , code can be generated with absolute addresses, but needs to be recompiled if the location changes
  - *Load time*: If a memory location is known *relative* to an address, code can be generated with relocatable addresses
  - *Execution time*: addresses are mapped dynamically during execution using *memory management* hardware

# *Multistep Processing of a User Program*

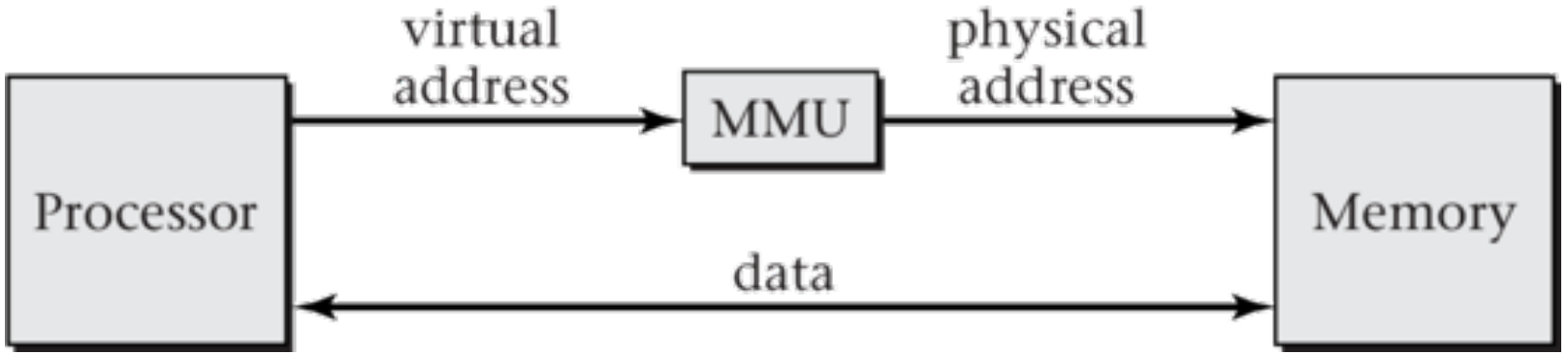


# *Logical vs. Physical Address Space*

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management
  - *Logical address (virtual address)* – generated by the CPU
    - *logical address space* is the set of all logical addresses generated
  - *Physical address* – address seen by the memory unit
    - *physical address space* is the set of all physical addresses used
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical and physical addresses MUST be different at execution time … Why?
  - Hardware-based address binding schemes here must do this
  - They must be fast!

# *Memory Management Unit*

- Hardware device that maps a logical (virtual) address to a physical address
- How it does this is at the heart of all memory management schemes
- Memory management unit (*MMU*) changes the logical address into a physical address



# *Contiguous Allocation*

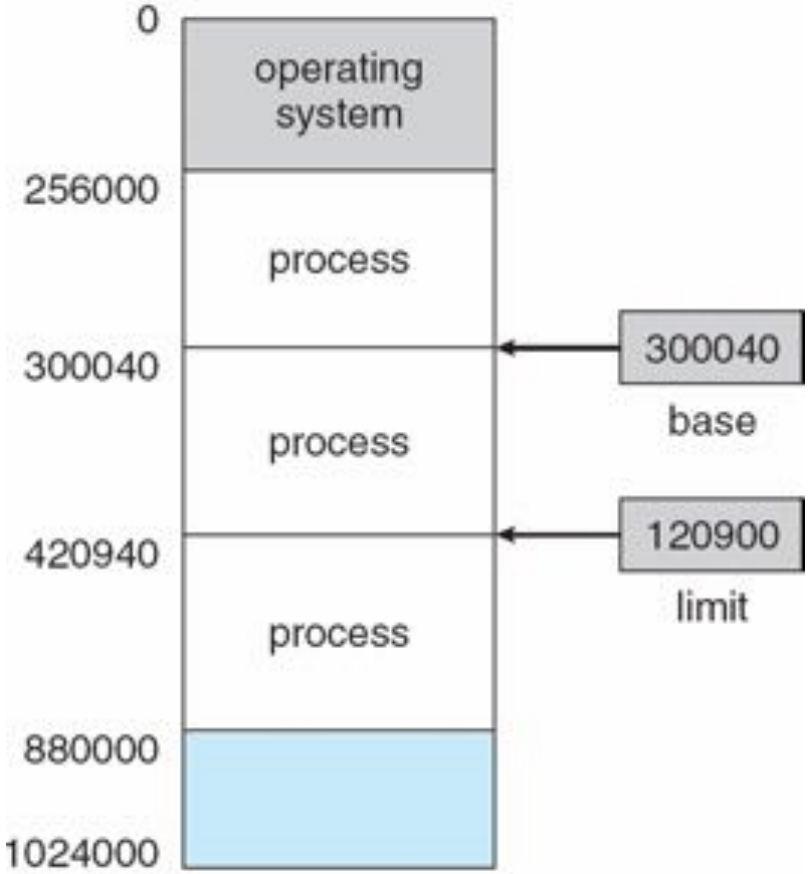
- Main (physical) memory must support both OS and user processes
  - It is a limited resource
  - Must allocated efficiently
- *Contiguous allocation* is one early method
- Main memory usually into two *partitions*:
  - Resident operating system
    - usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section (partition) of memory

# *Contiguous Allocation Address Translation*

- One mechanism is to offset all logical address
  - A *base register* contains a value that is added to every logical address before it is sent to memory
  - A *limit register* is use to set a bound on the maximum address
- Another mechanism is to *relocate* an address
  - One or more relocation register(s) contains a value that is added to every logical address of a memory region before it is sent to memory
- User program's generate logical addresses
  - Their logical addresses change to physical addresses

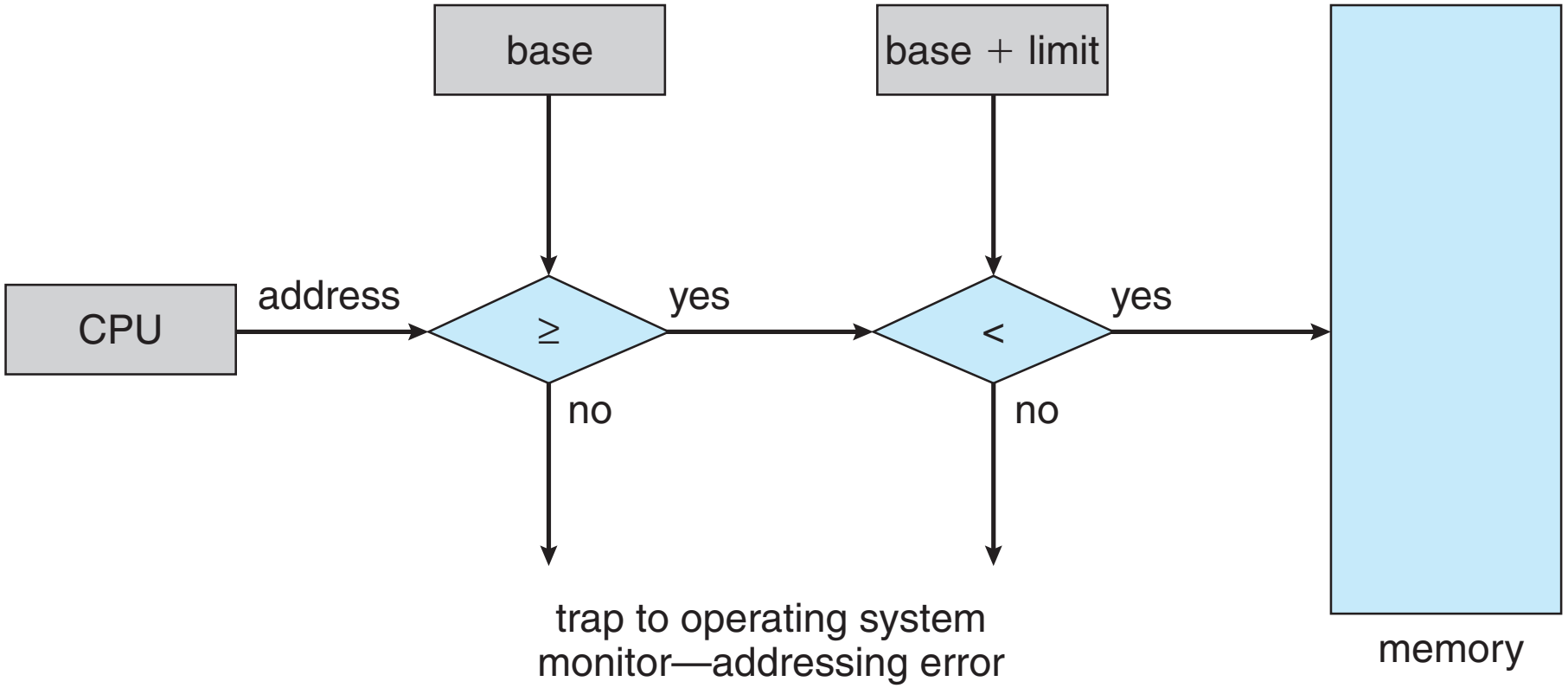
# *Base and Limit Registers with Partitioning*

- How do we support the *logical address space*
  - A pair of *base* and *limit registers*
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
  - If not, an exception occurs
- Need hardware support for doing this
- Useful for mapping entire process address space into a fixed address region
- OS sets up base/limit values



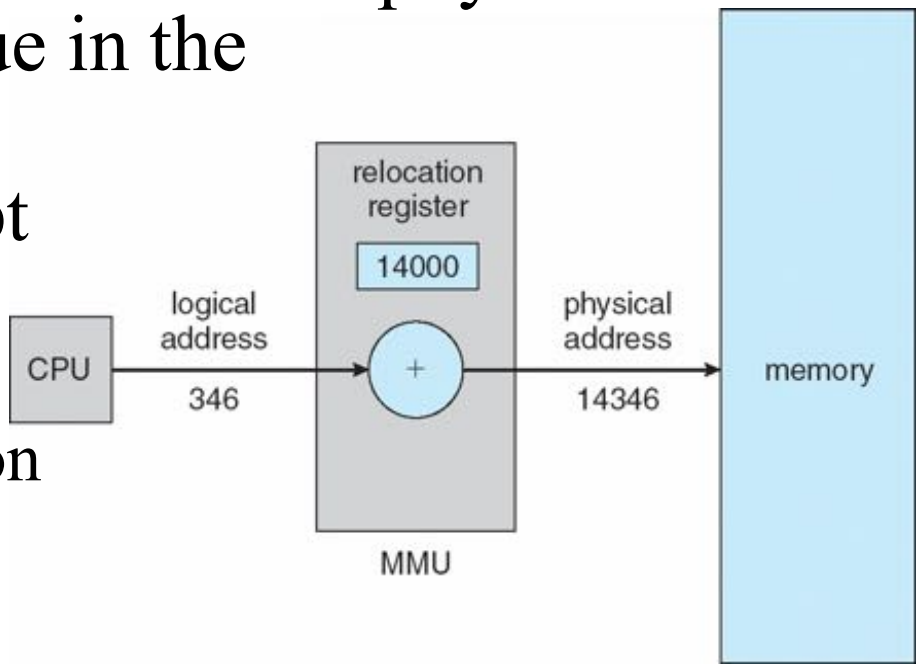
# *Hardware Address Protection*

- If the OS did the check, it would be slow
- Hardware is used instead



# *Dynamic Relocation via Relocation Register*

- CPU logical address is converted to a physical address by adding the value in the relocation register
- Like a base register, except you can have multiple relocation registers
  - One for each memory region
  - Code region
  - Heap region
  - Different routines (dynamic loading)
- Little support from the operating system is required
  - Implemented through program design
  - Used in coordination with base/limit registers

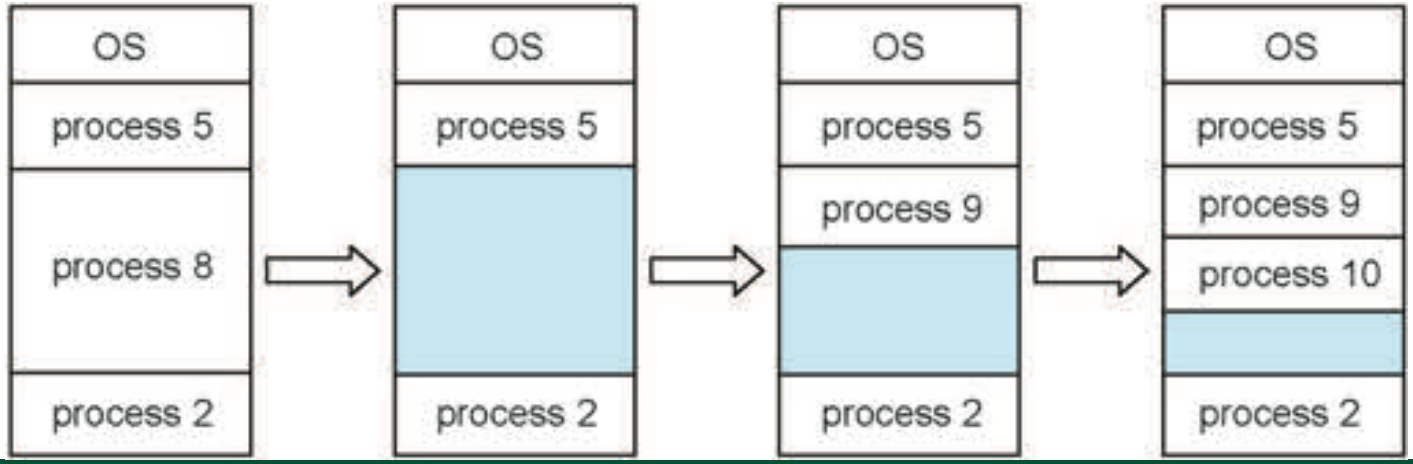


# *Dynamic Linking*

- *Static linking* – system libraries and program code combined by the loader into the binary program image
- *Dynamic linking* –linking postponed until execution time
- Small piece of code (stub) used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine and executes the routine
- OS checks if routine is in the processes memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries (Why?)
  - System libraries loaded when the program starts are also known as *shared libraries*
- Consider dynamic linking use to patching system libraries
  - Versioning may be needed

# *Multiple-partition Allocation*

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - Variable-partition sizes for efficiency (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
  - Operating system maintains information about:
    a) allocated partitions  b) free partitions (*hole*)



# *Dynamic Storage Allocation Problem*

- How to satisfy a request of size *n?*
  - Assume there is a list of free holes
- *First-fit*
  - Allocate the first hole that is big enough
- *Best-fit*
  - Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- *Worst-fit*
  - Allocate the largest hole; must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

# *Fragmentation*

- *External fragmentation*
  - *T*otal memory space exists to satisfy a request, but it is not contiguous
- *Internal fragmentation*
  - *A*llocated 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*

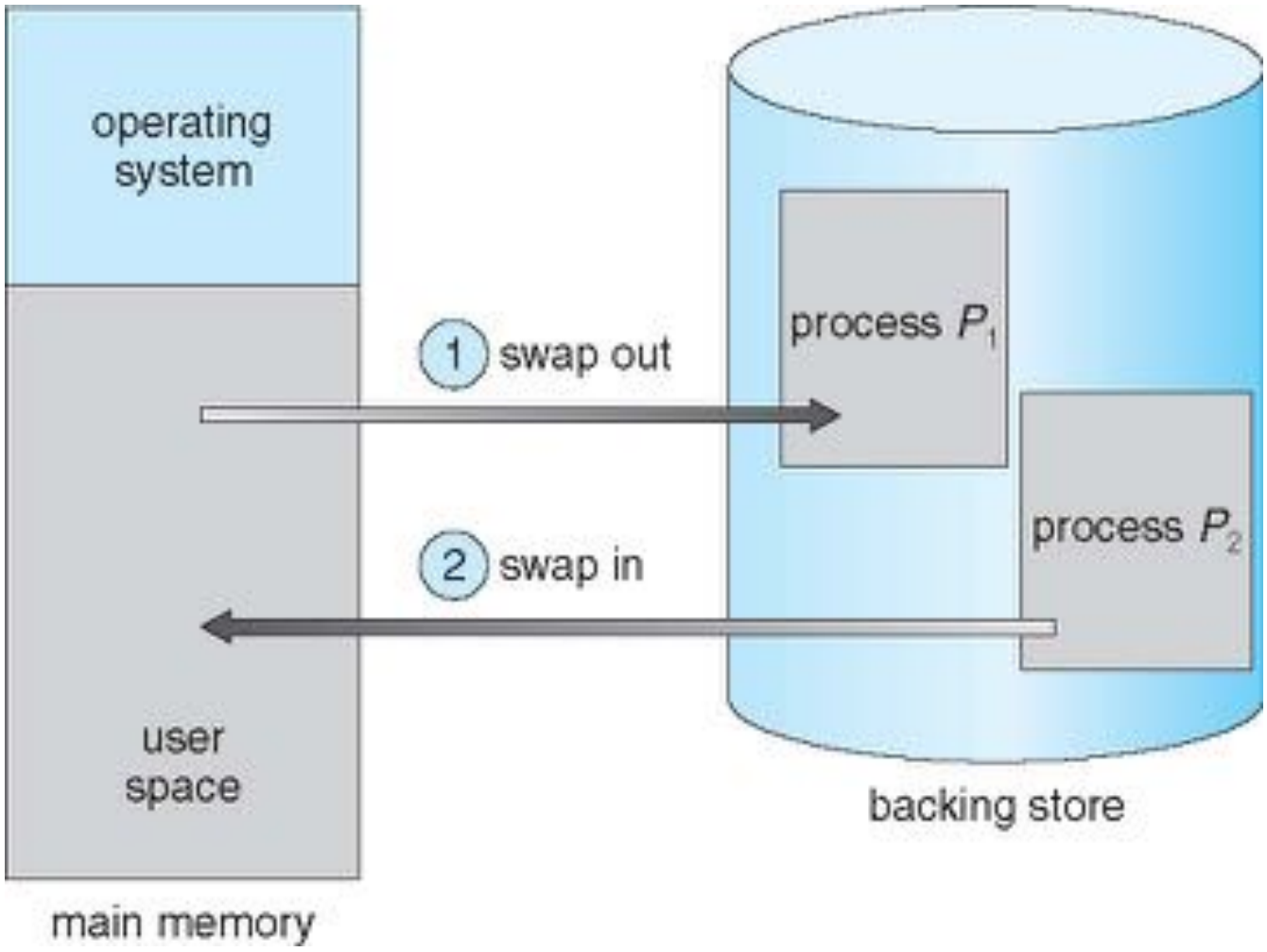
# *Fragmentation Improvement*

- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem
    - keep job in memory while it is involved in I/O
    - do I/O only into OS buffers

# *Swapping*

- Physical memory space is finite
- OS takes up some portion of the physical memory
- Suppose we allocate physical memory to meet the needs of a process as discussed
  - What happens when the total physical memory space requested by processes exceeds physical memory?
  - Just consider 2 processes
- Swapping is the idea of sharing physical memory space by having only 1 process use the space at any time
- Still think about this with respect to contiguous memory allocation for a process

# *Schematic View of Swapping*



# *Swapping Support*

- A process can be *swapped* temporarily out of memory
  - Moved to a *backing store*– fast disk large enough to accommodate copies of all memory images for all users
  - Later brought back into memory for continued execution
- Swapping is used along with process scheduling
  - *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms
  - Lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time
  - Total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk

# *Swapping Methods*

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Also consider pending I/O since it might be associated with particular addresses
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if greater than a memory allocation threshold
  - Disabled again once memory demand reduced

# *Context Switch Time including Swapping*

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
  - 100MB process swapping to hard disk
  - 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
- Can reduce if reduce size of memory swapped
  - Know how much memory really being used
  - System calls to inform OS of memory use via *request_memory()* and *release_memory()*

# *Swapping Constraints*

- Other constraints as well on swapping
  - Can not swap out if process has pending I/O
    - otherwise I/O would occur to wrong process
  - Could transfer I/O to kernel space, then to I/O device
    - known as *double buffering*, adds overhead
- Standard swapping not really used in modern operating systems
  - But modified version common
    - swap only when free memory extremely low

# *Swapping on Mobile Systems*

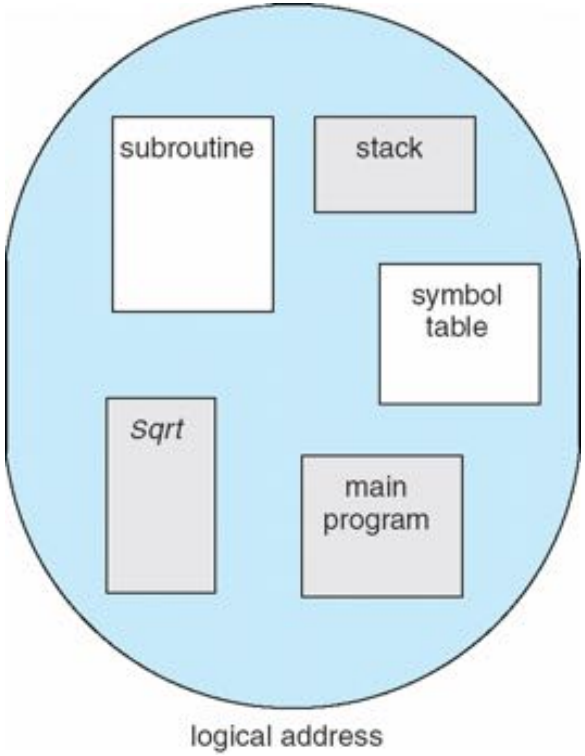
- Not typically supported
  - Flash memory based
    - small amount of space
    - limited number of write cycles
    - poor throughput between flash memory and CPU for mobile
- Instead use other methods to free memory if low
  - iOS asks apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes application state to flash for fast restart
  - Both OSes support paging as discussed below

# *Non-contiguous Memory Allocation*

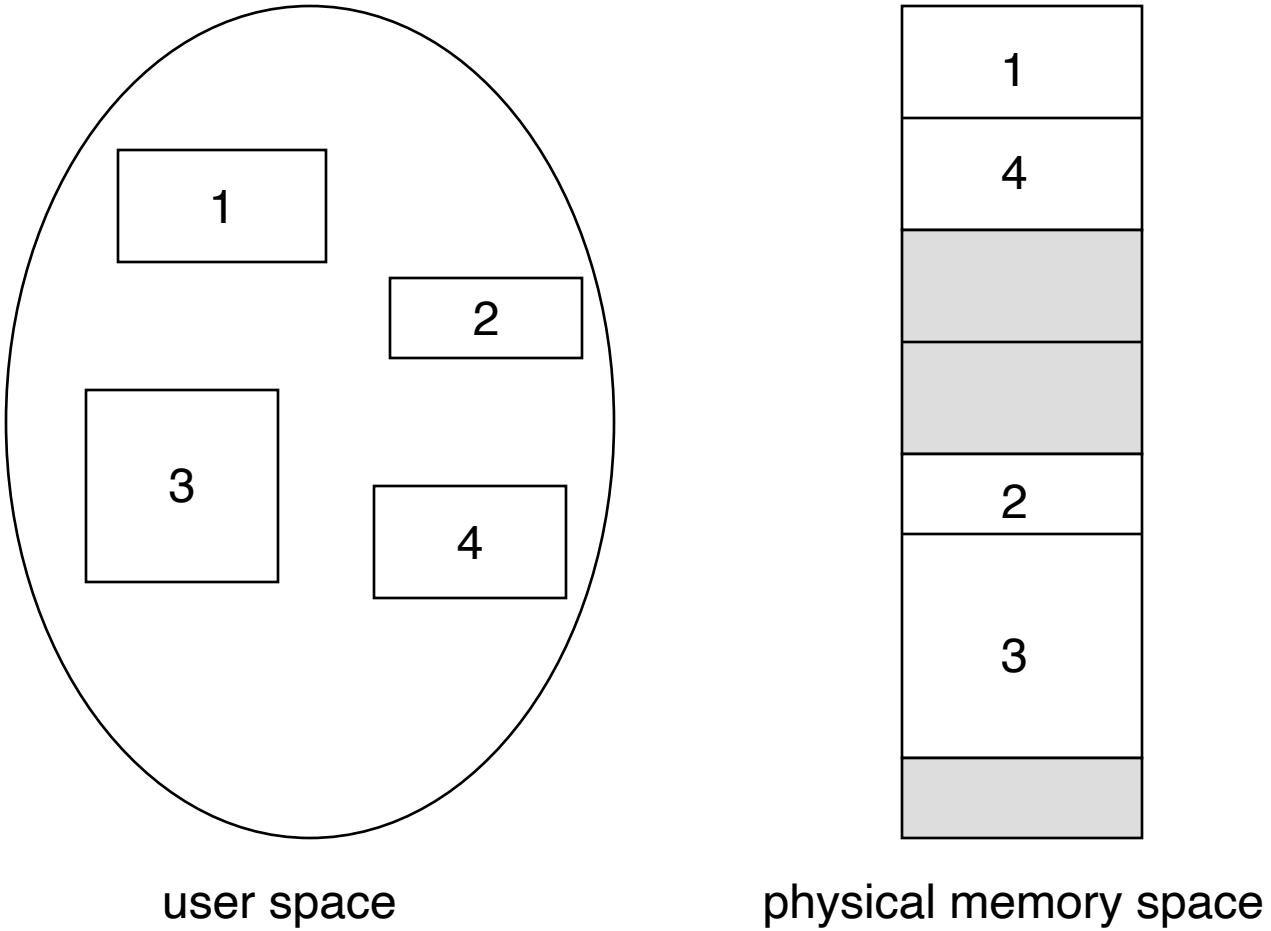
- Contiguous memory allocation is not all that great for managing memory
- Its advantages are that address translation is very straightforward and fast
  - Just need a base and limit register
- However, there are disadvantages
  - All logical addresses must be contiguous
  - Holes can develop requiring compaction to fix
- Suppose we loosen constraints and allow for non-contiguous memory allocation
  - Segments
  - Pages

# *Segmentation*

- Memory-management scheme that supports a user view of memory
  - Allows different parts to be allocated separately
- A program is a collection of segments
  - A segment is a logical unit
  - Instructions
    - main program
    - procedures and functions
    - libraries
  - Data
    - global variables
    - stack
    - arrays
- Associate (continuous) logical addresses with each segment



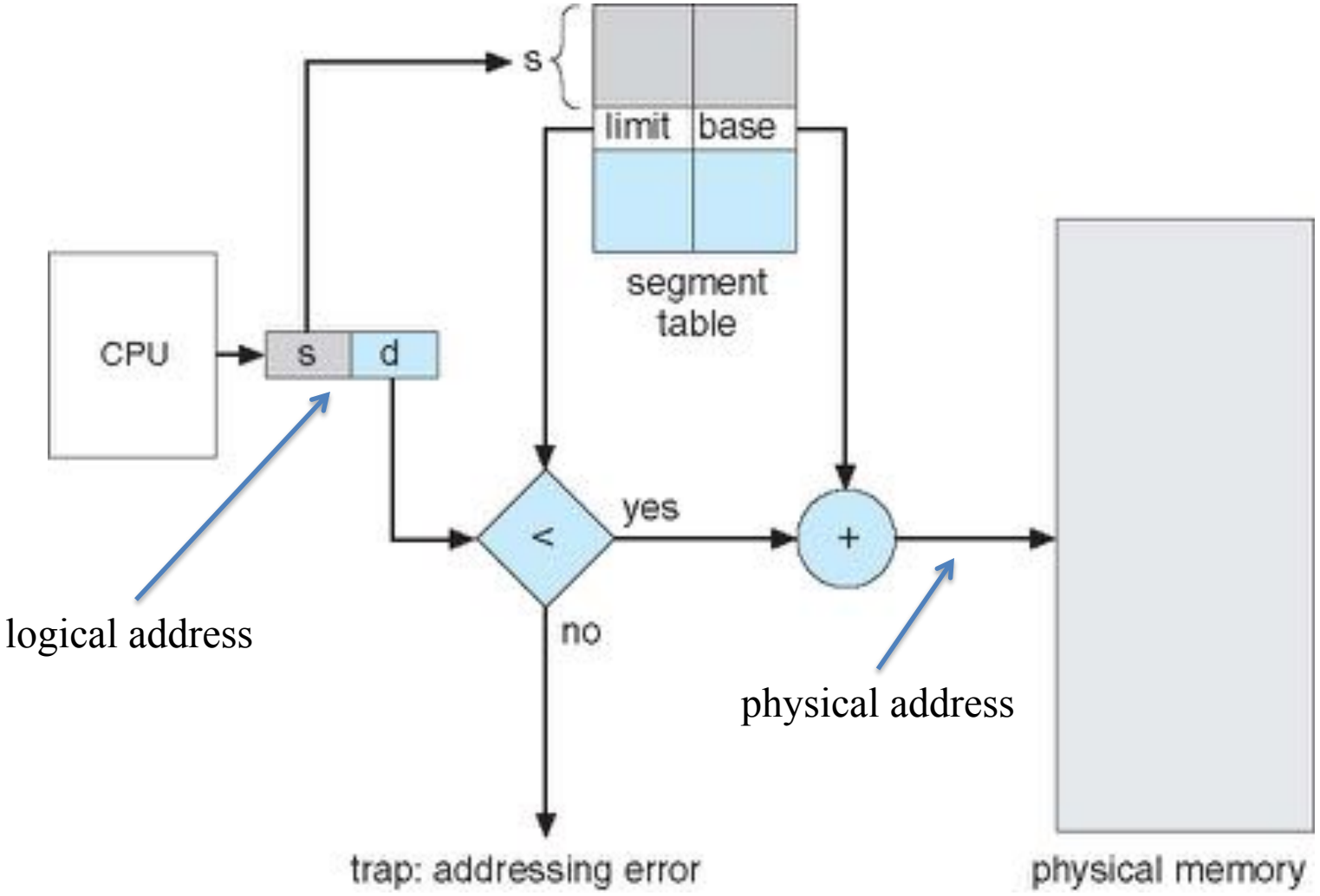
# *Logical View of Segmentation*



# *Segmentation Architecture*

- Logical address consists of a two tuple: *<segment-number, offset>*
- *Segment table*
  - *M*aps two-dimensional physical addresses
  - Each table entry has:
    - *base register* containing the starting physical address where the segments reside in memory
    - *Limit register* specifying the length of the segment
- *Segment-table base register (STBR)* points to the 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

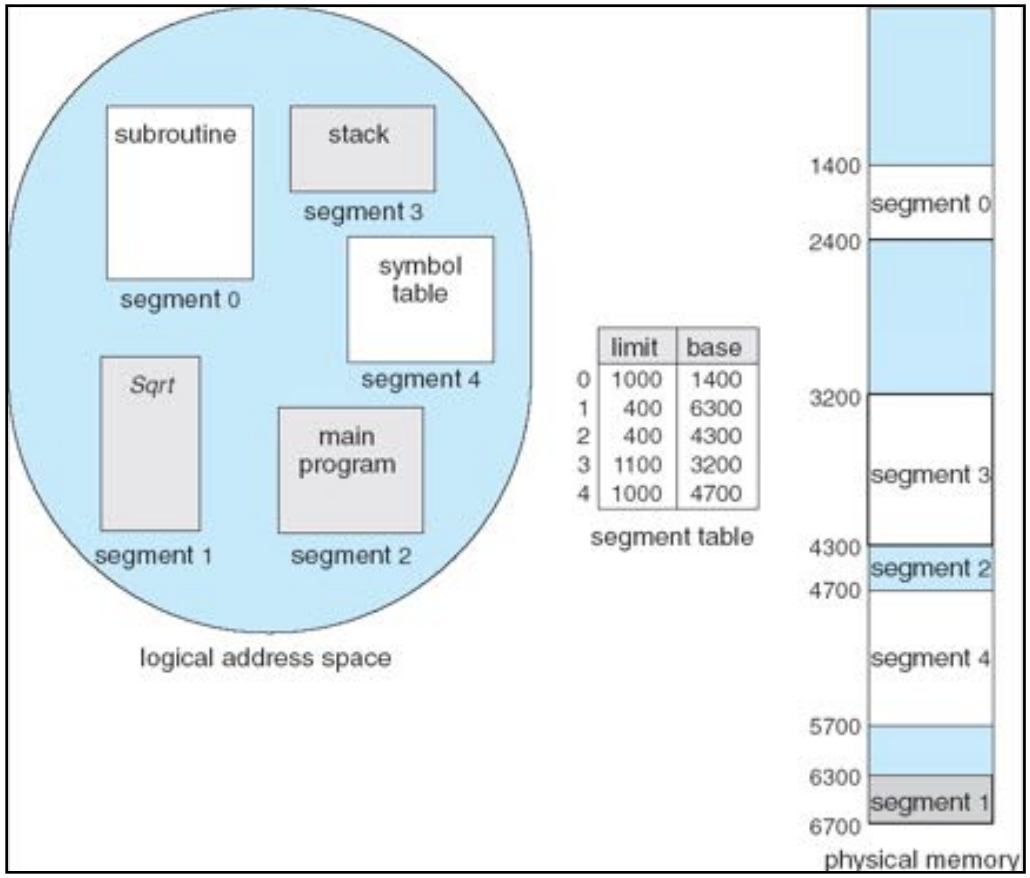
# Segmentation Hardware

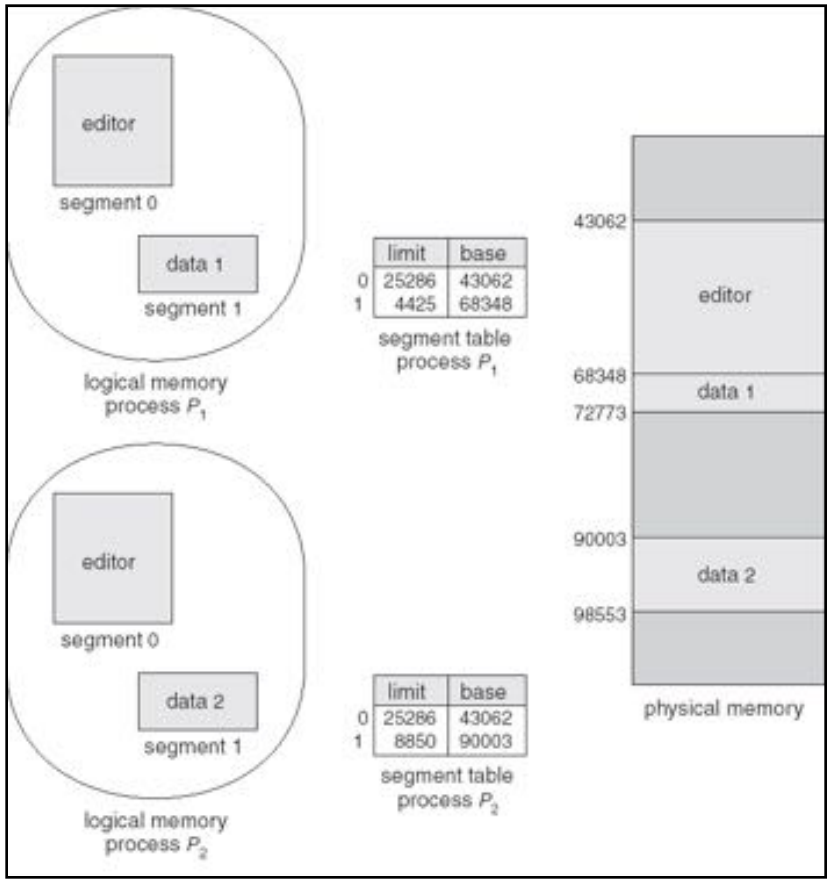


# *Segmentation Architecture Protection*

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments
  - Code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
  - Could possibly still result in areas of physical memory that are too small to be allocated to a segment (holes)
- A segmentation example is shown next

# Segmentation and Segment Sharing Example



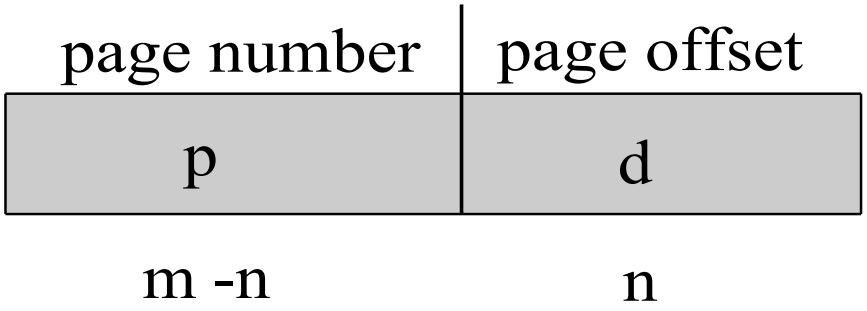



# *Paging*

- Suppose we want to eliminate external fragmentation entirely
- 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*
- 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
- Physical address space of a process is noncontiguous
- Memory management reduces to finding frames for pages
  - Process is allocated physical memory wherever pages are available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Backing store likewise split into pages!
- Still have internal fragmentation … Why?

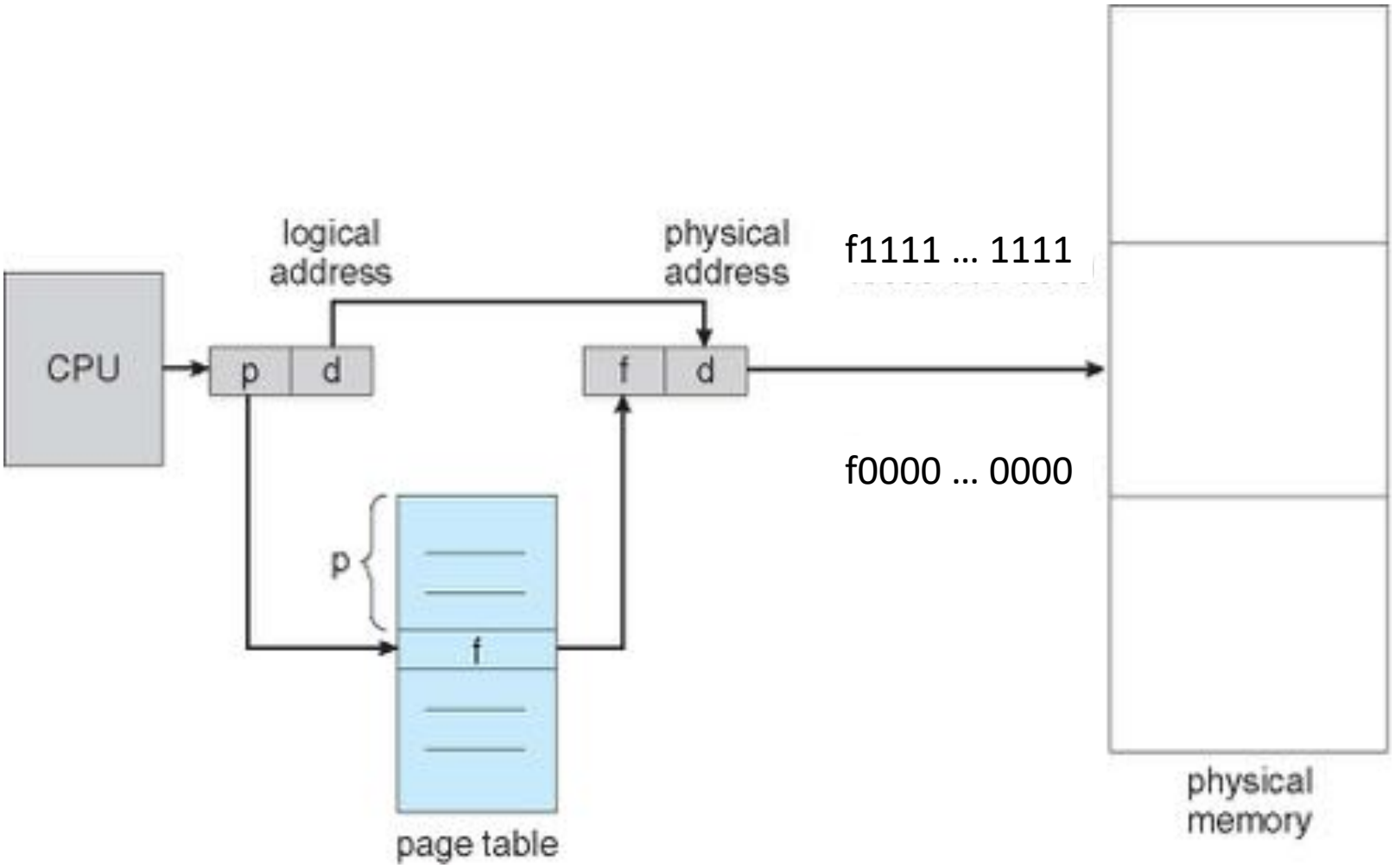
# *Address Translation Scheme*

- 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

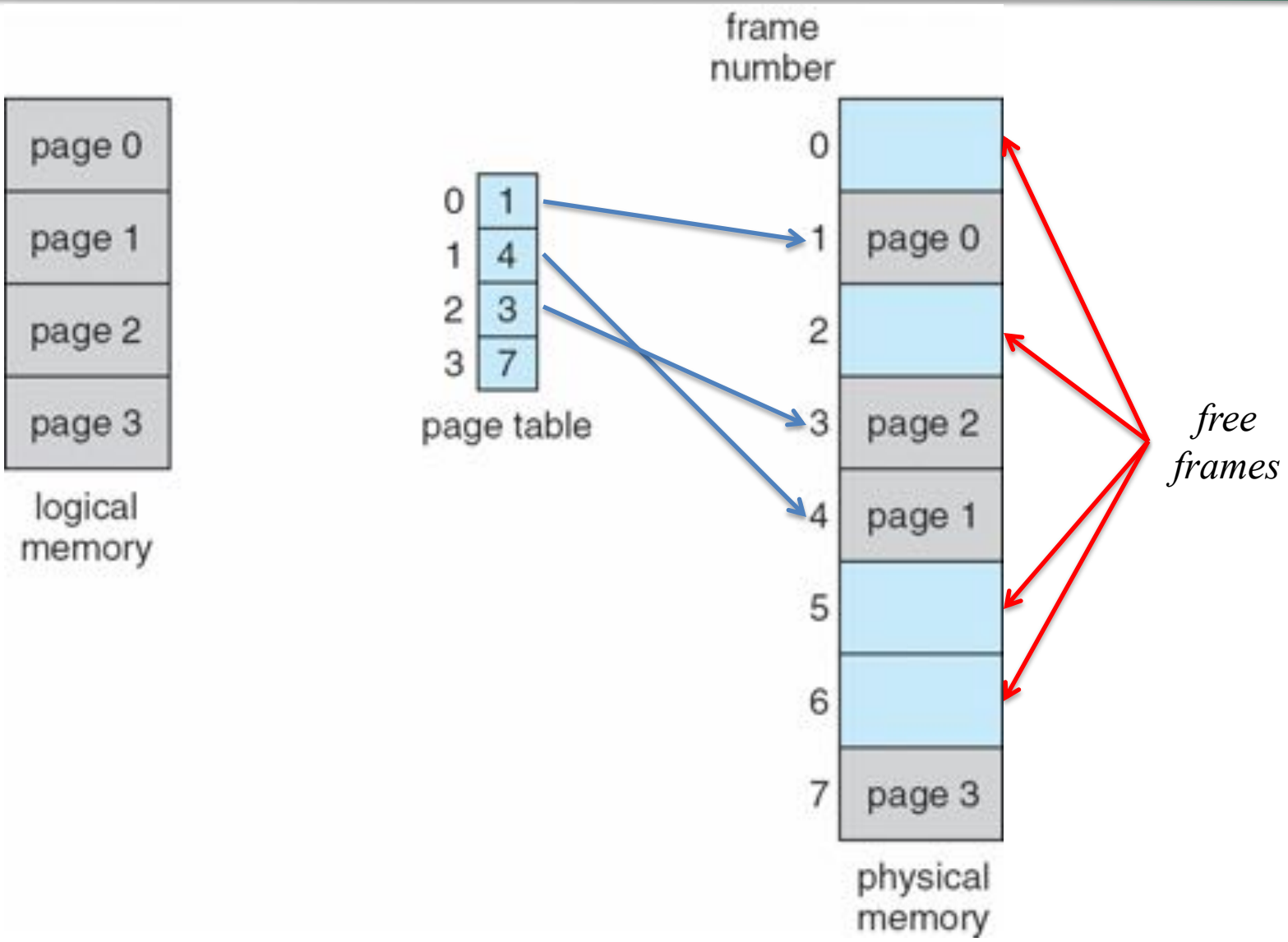



  - For given logical address space $2^m$ and page size $2^n$

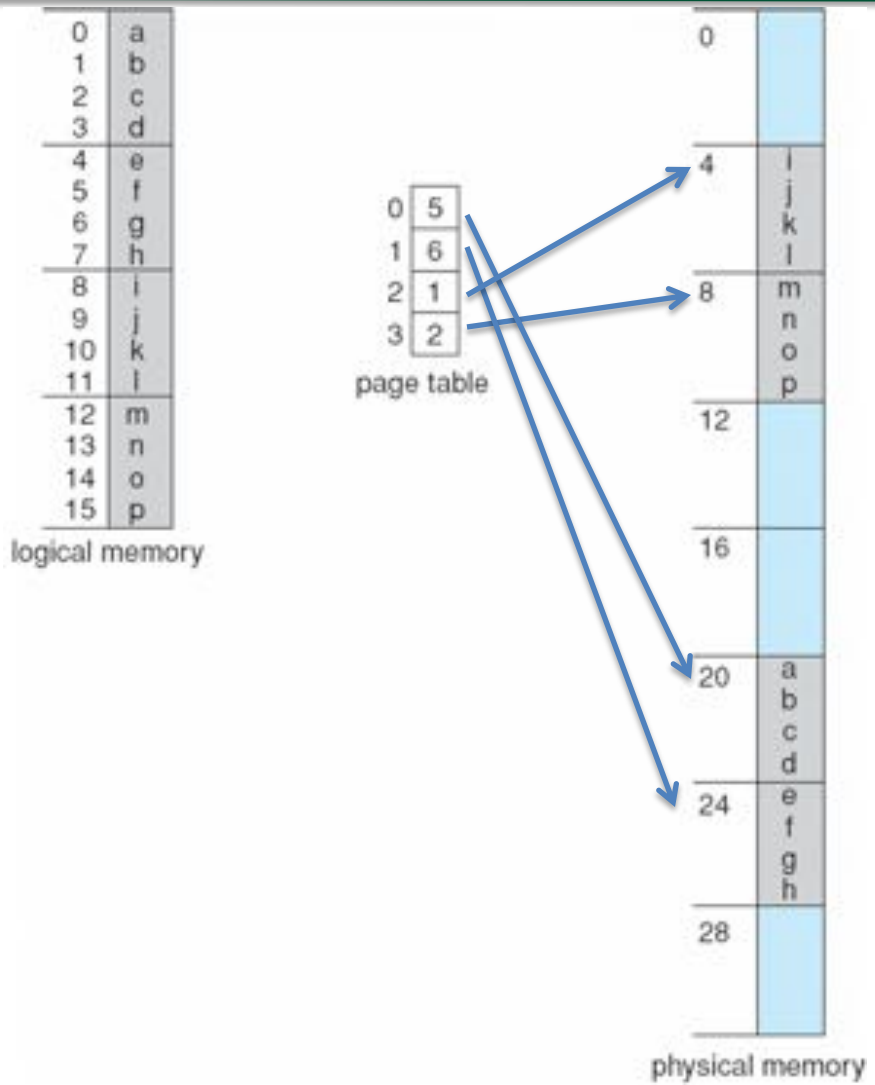
# Paging Hardware



# *Paging Model of Logical / Physical Memory*



# *Paging Example*

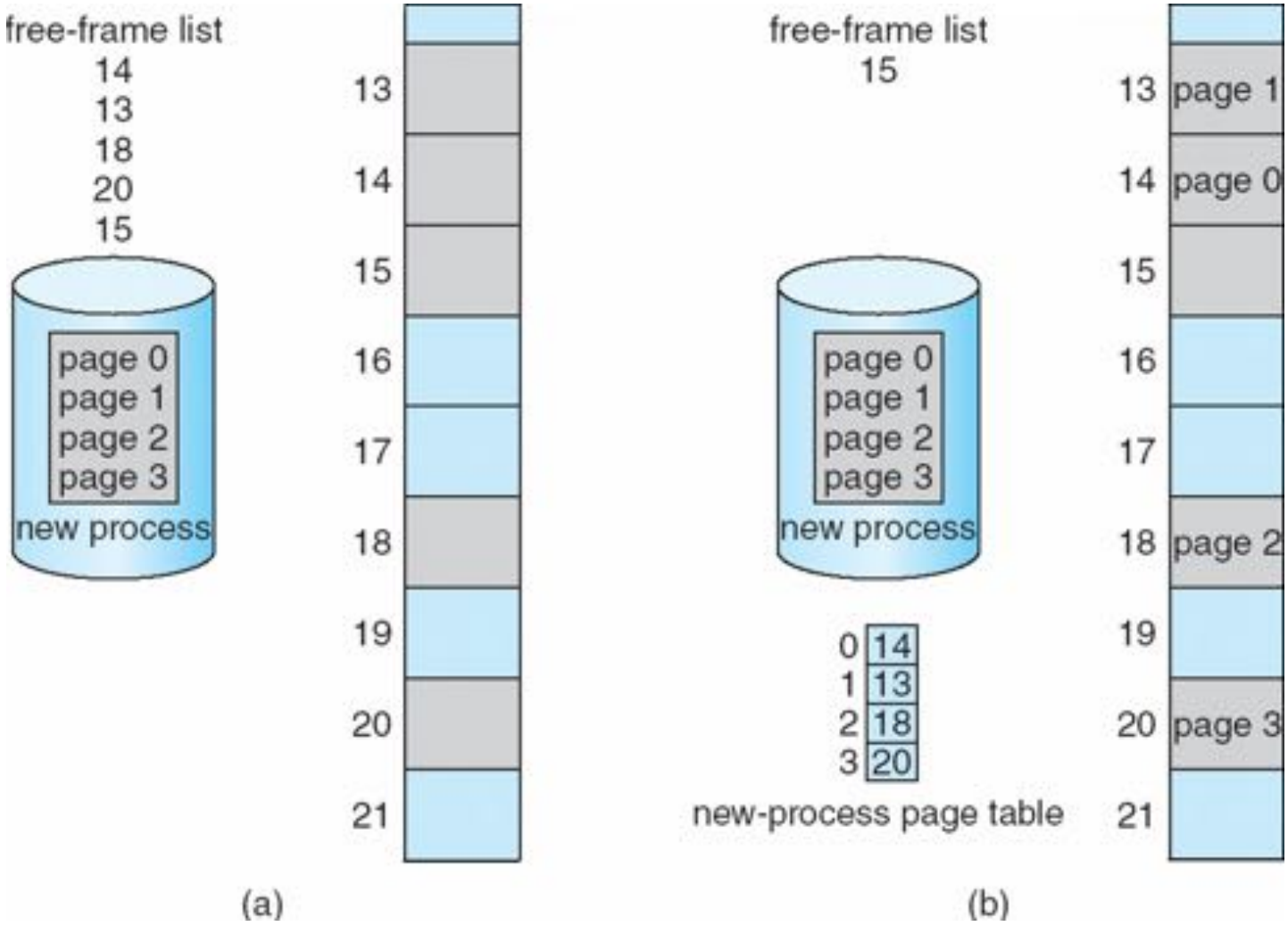



*n*=2 and *m*=4  32-byte memory and 4-byte pages

# *Paging and Fragmentation*

- 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
- Two memory accesses can be improved by the use of a special fast-lookup hardware cache called *translation look-aside buffers (TLBs)*
  - Implemented with associative memory

# *Page Table TLB*

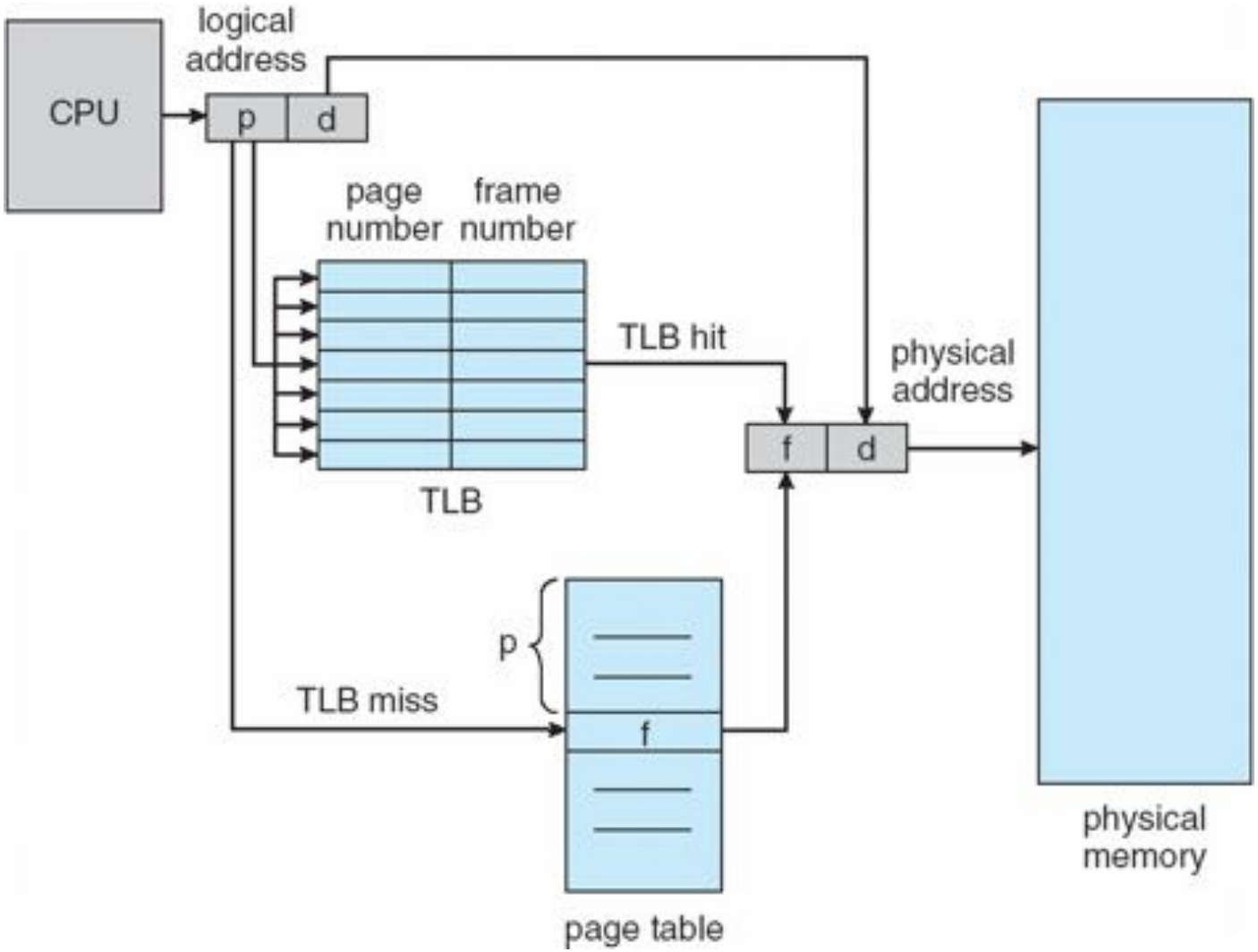
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be wired down for permanent fast access
- Some TLBs store *address-space identifiers (ASIDs)* in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch

# *Associative Memory*

- Associative memory – parallel search


- 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 = 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
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- *Effective Access Time (EAT)*

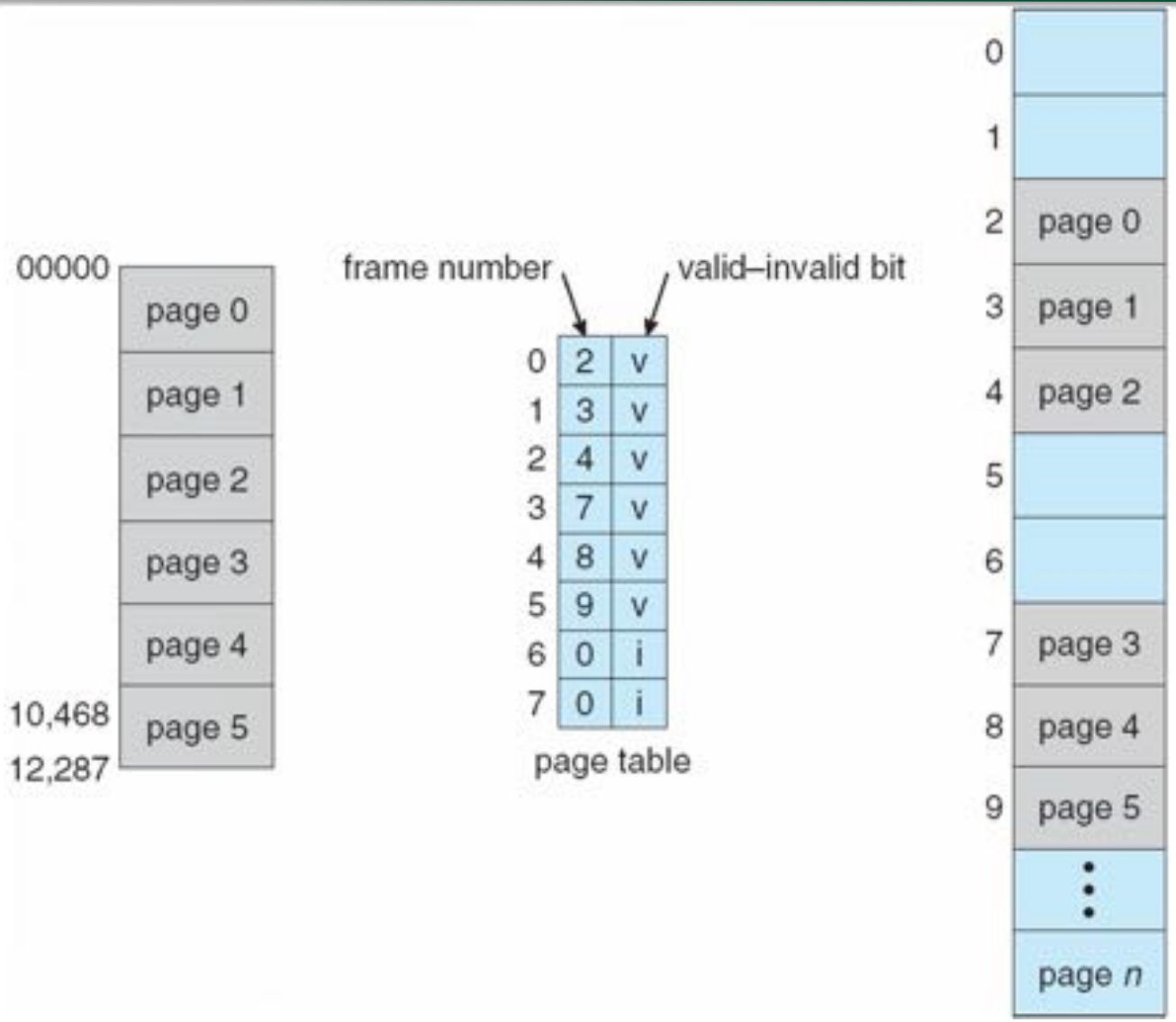
$$\begin{aligned} EAT &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- 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, and so on
- Use a *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

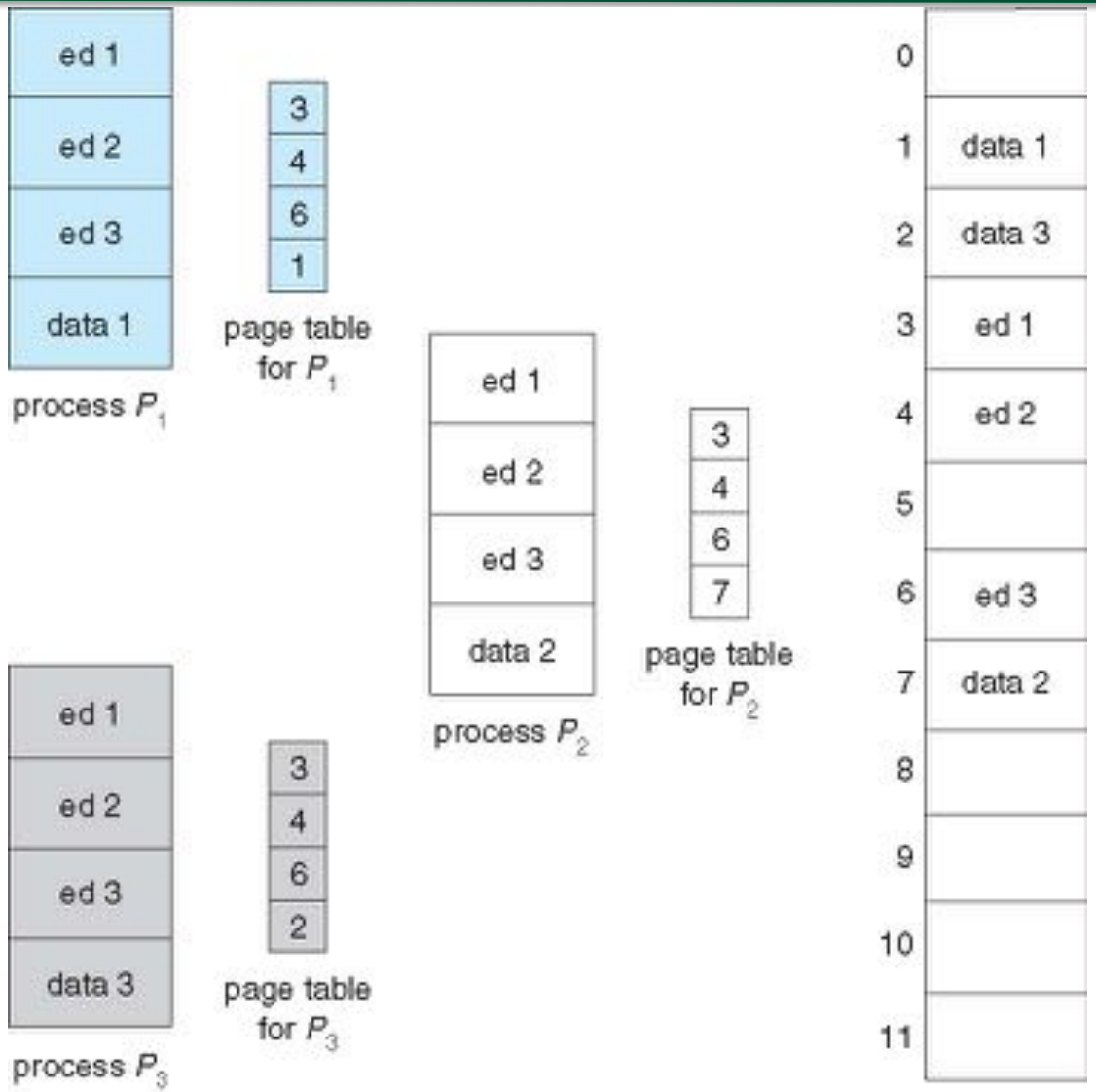
# *Valid (v) or Invalid (i) Bit In A Page Table*



# *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*



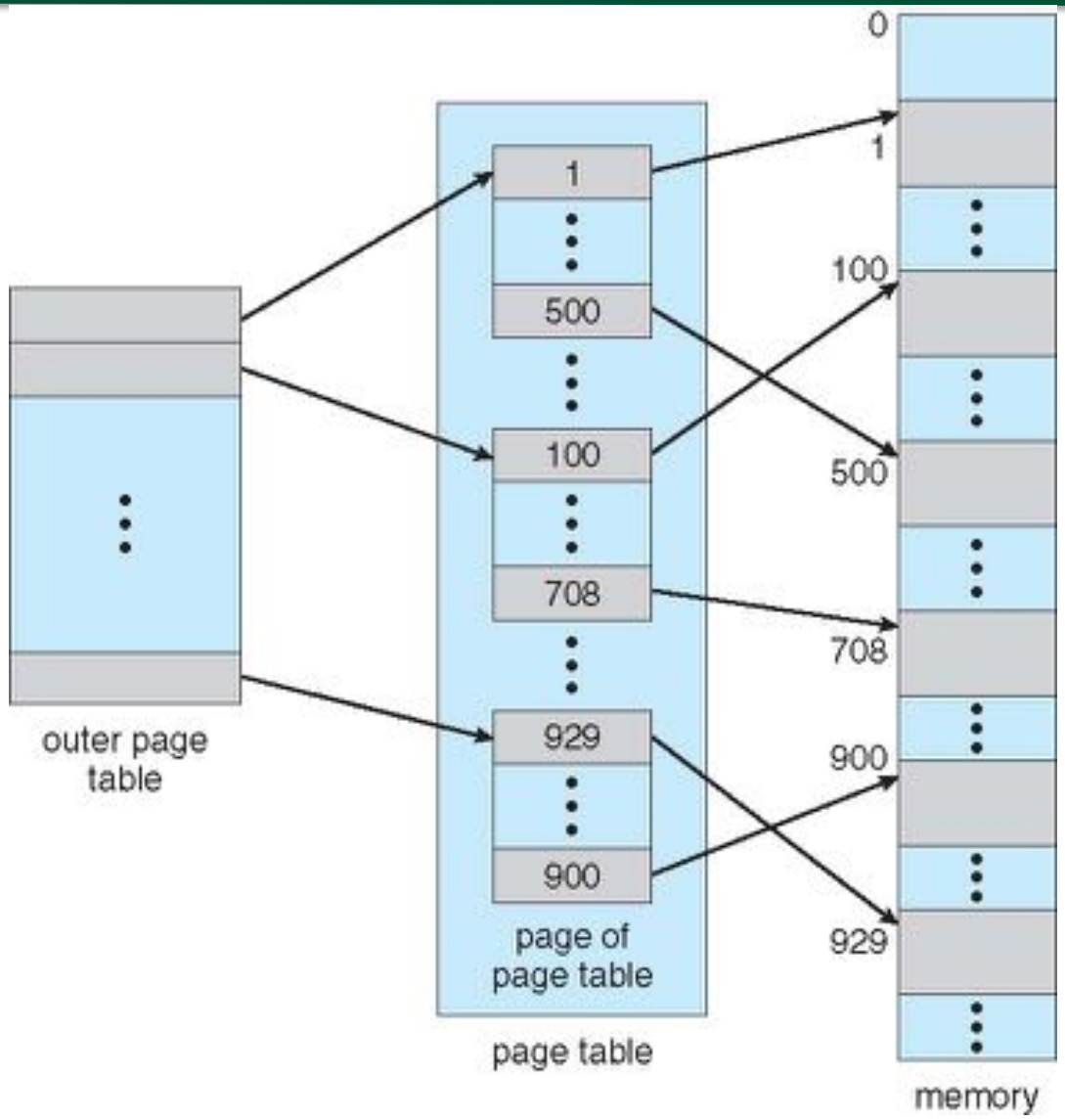
# *Structure of the Page Table*

- 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
    - do not want to allocate that contiguously in main memory
- Hierarchical paging
- Hashed page tables
- Inverted page tables

# *Hierarchical Page Tables*

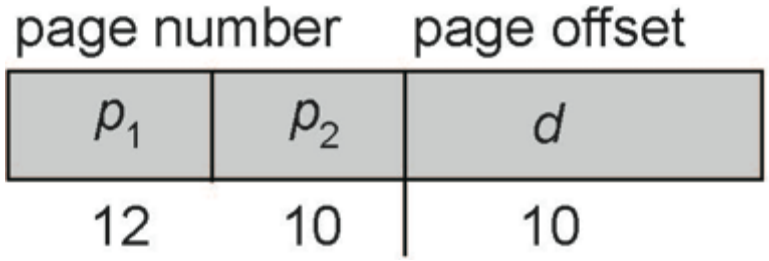
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

# *Two-Level Page-Table Scheme*



# *Two-Level Paging Example*

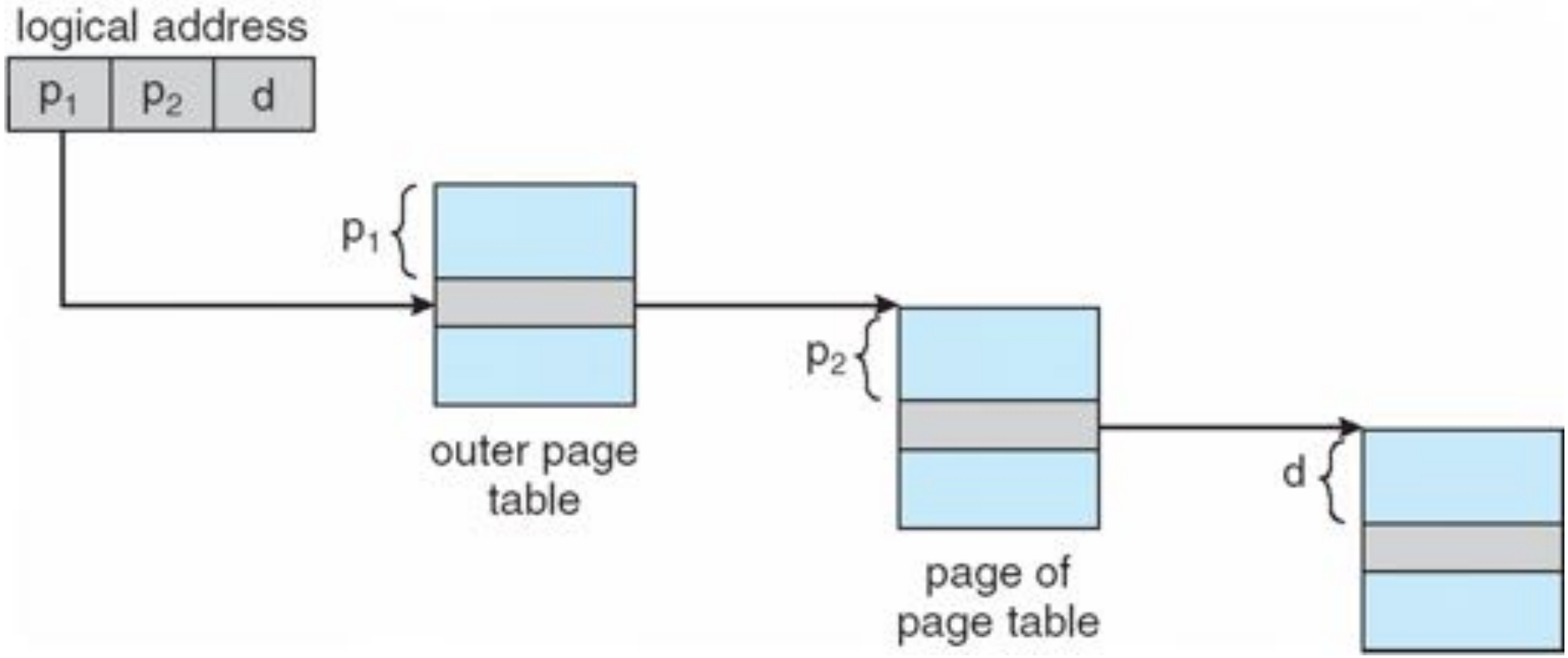
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - A page number consisting of 22 bits
  - A page offset consisting of 10 bits
- Since the page table is paged, the page number is divided into:
  - 12-bit page number
  - 10-bit page offset
- Thus, a logical address is as follows:




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

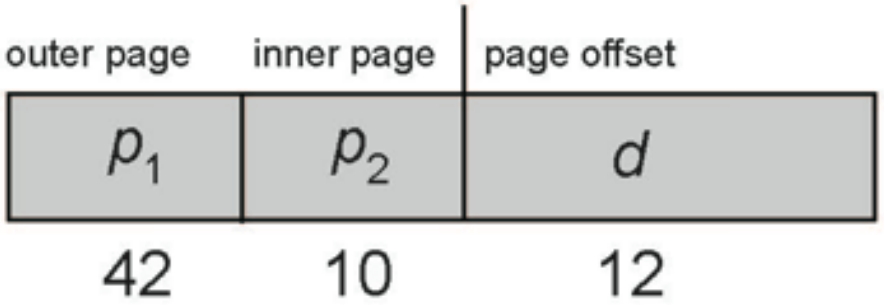
- Known as *forward-mapped page table*

# *Address-Translation Scheme*



# 64-bit Logical Address Space

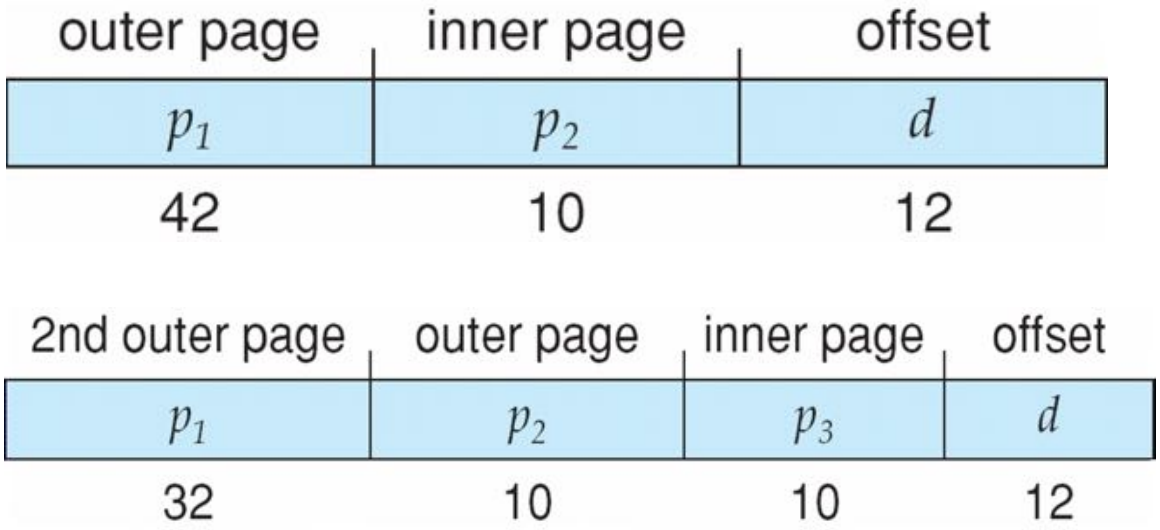
- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
  - Then page table has $2^{52}$ entries
  - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  - Address would look like




  - Outer page table has $2^{42}$ entries or $2^{44}$ bytes

# 2nd Outer Page Table

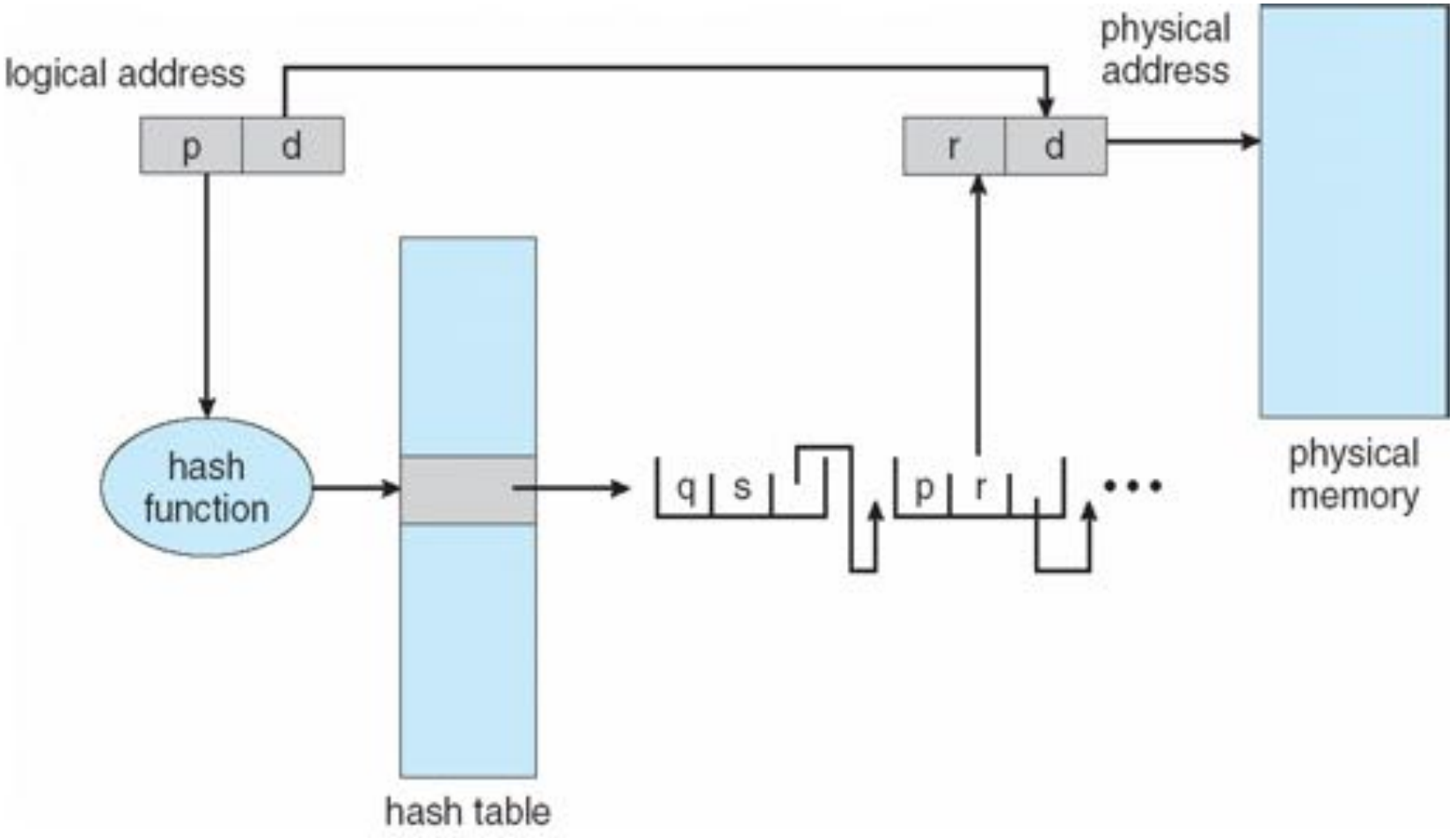
- One solution is to add a 2nd outer page table
- In this example, the 2nd outer page table is still 234 bytes in size
  - Possibly 4 memory access to get to one physical memory location



# *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)
- Hashing has to be done by the MMU … Why?

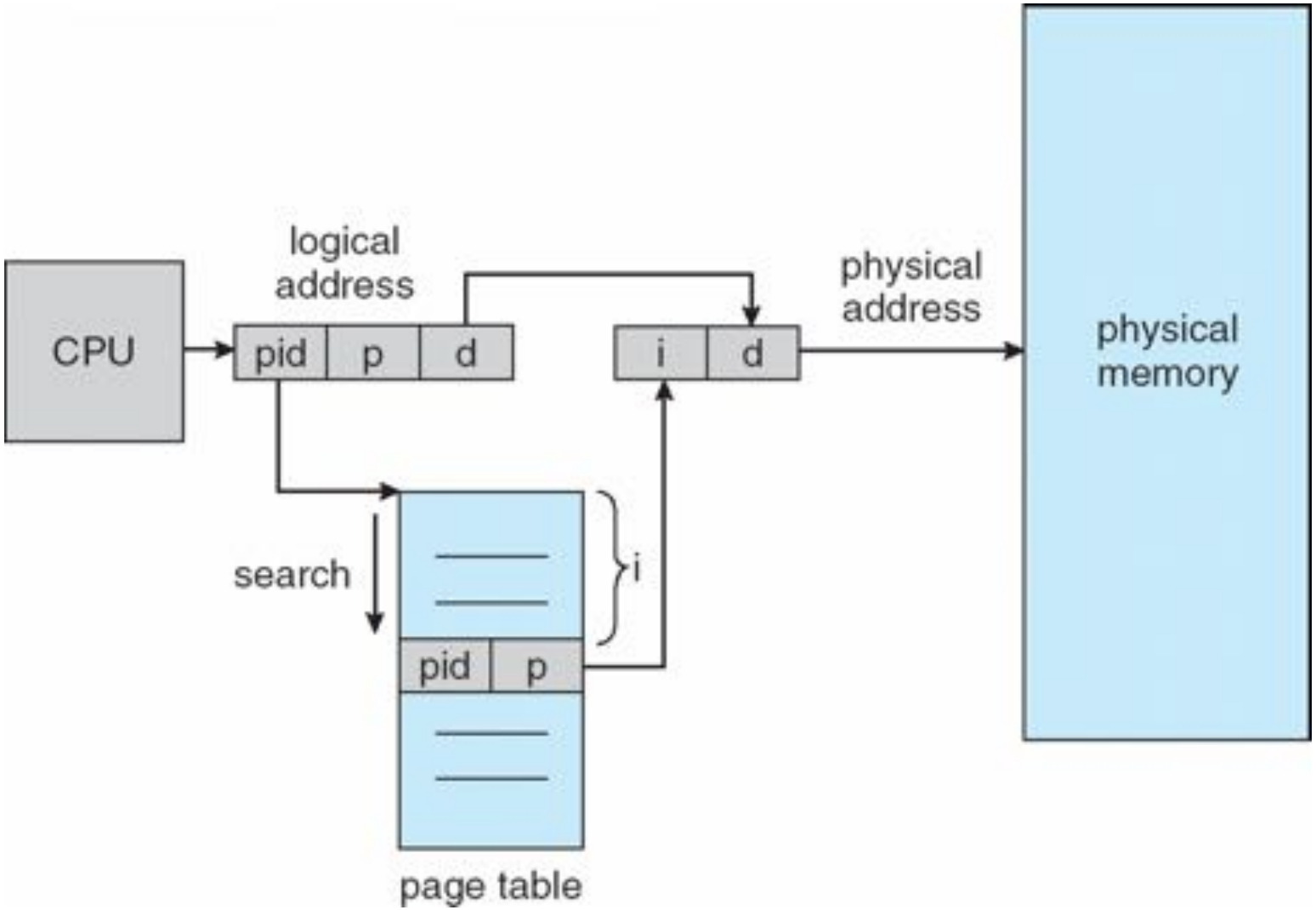
# Hashed Page Table



# *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*



# *The Intel 32 and 64-bit Architectures*

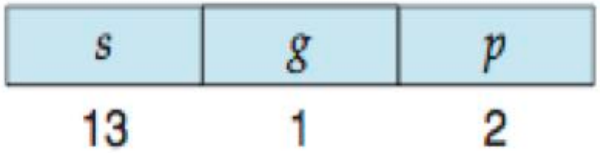
- Dominant industry chips
- Pentium CPUs are 32-bit
  - Called *IA-32* architecture
- Current Intel CPUs are 64-bit
  - Called *IA-64* architecture
- Many variations in the chips, cover the main ideas here

# *The Intel IA-32 Architecture*

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K 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)*)

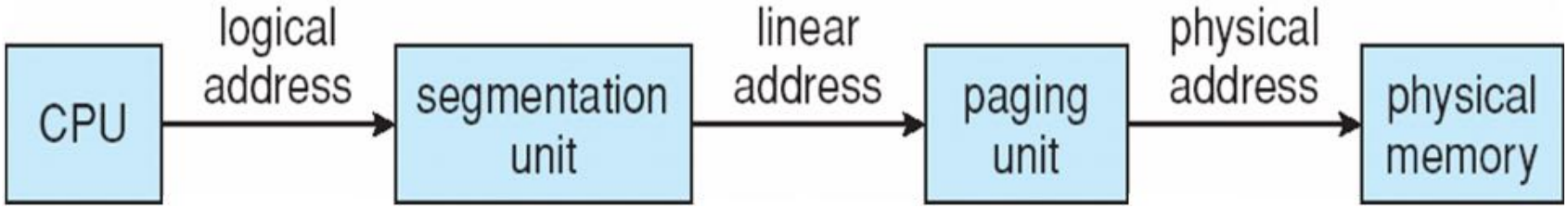
# *The Intel IA-32 Architecture Addresses*

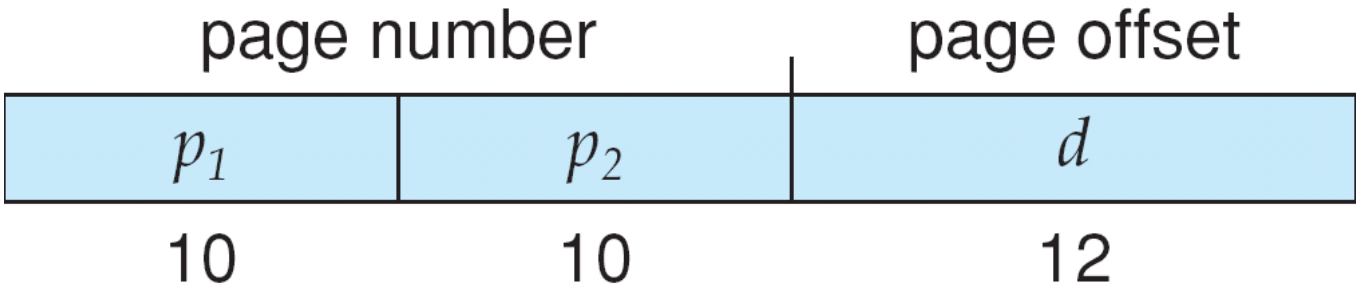
- CPU generates logical address
  - Selector given to segmentation unit
    - which produces linear addresses



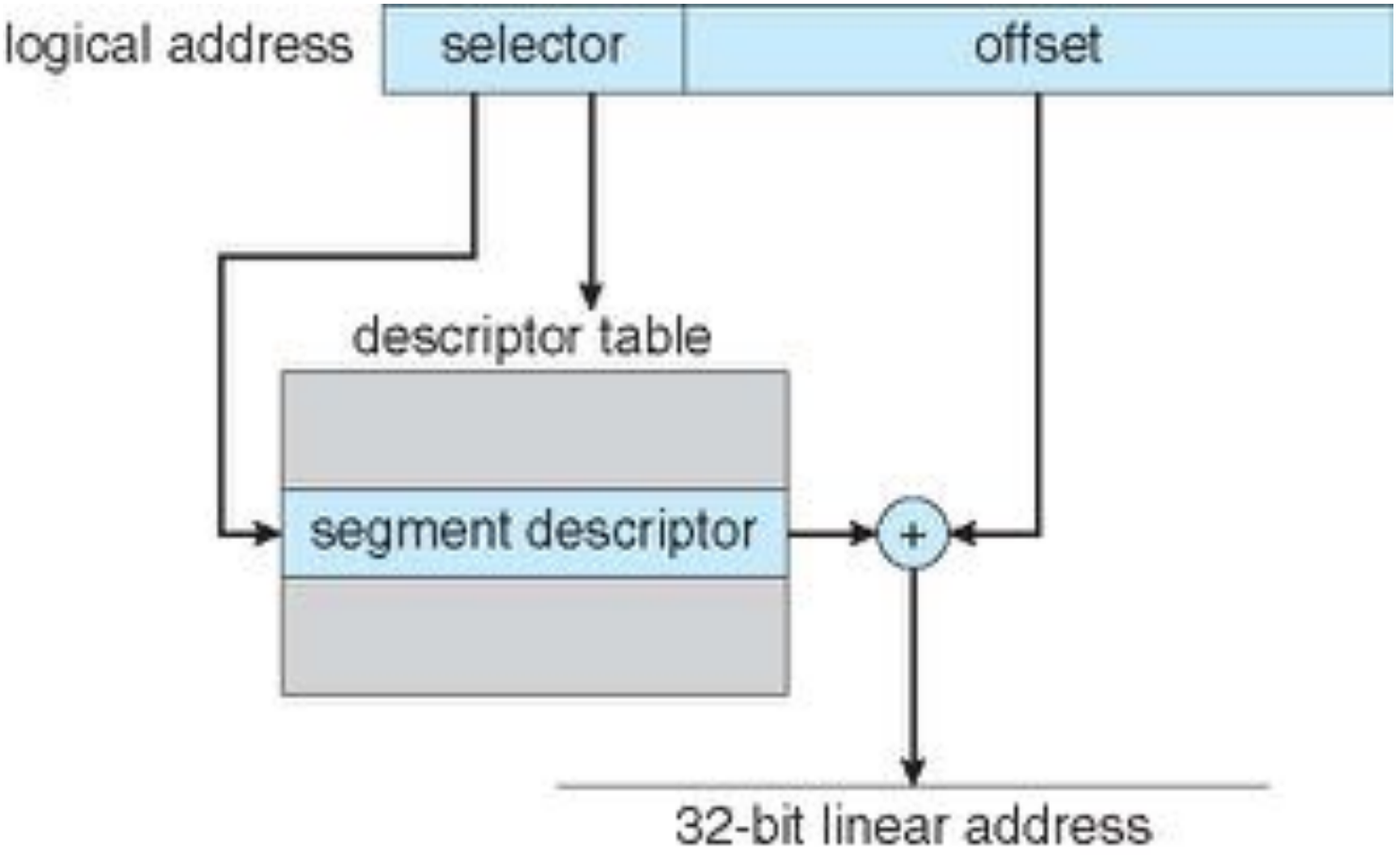

  - Linear address given to paging unit
    - which generates physical address in main memory
    - paging units form equivalent of MMU
    - pages sizes can be 4 KB or 4 MB

# IA-32 Logical to Physical Address Translation

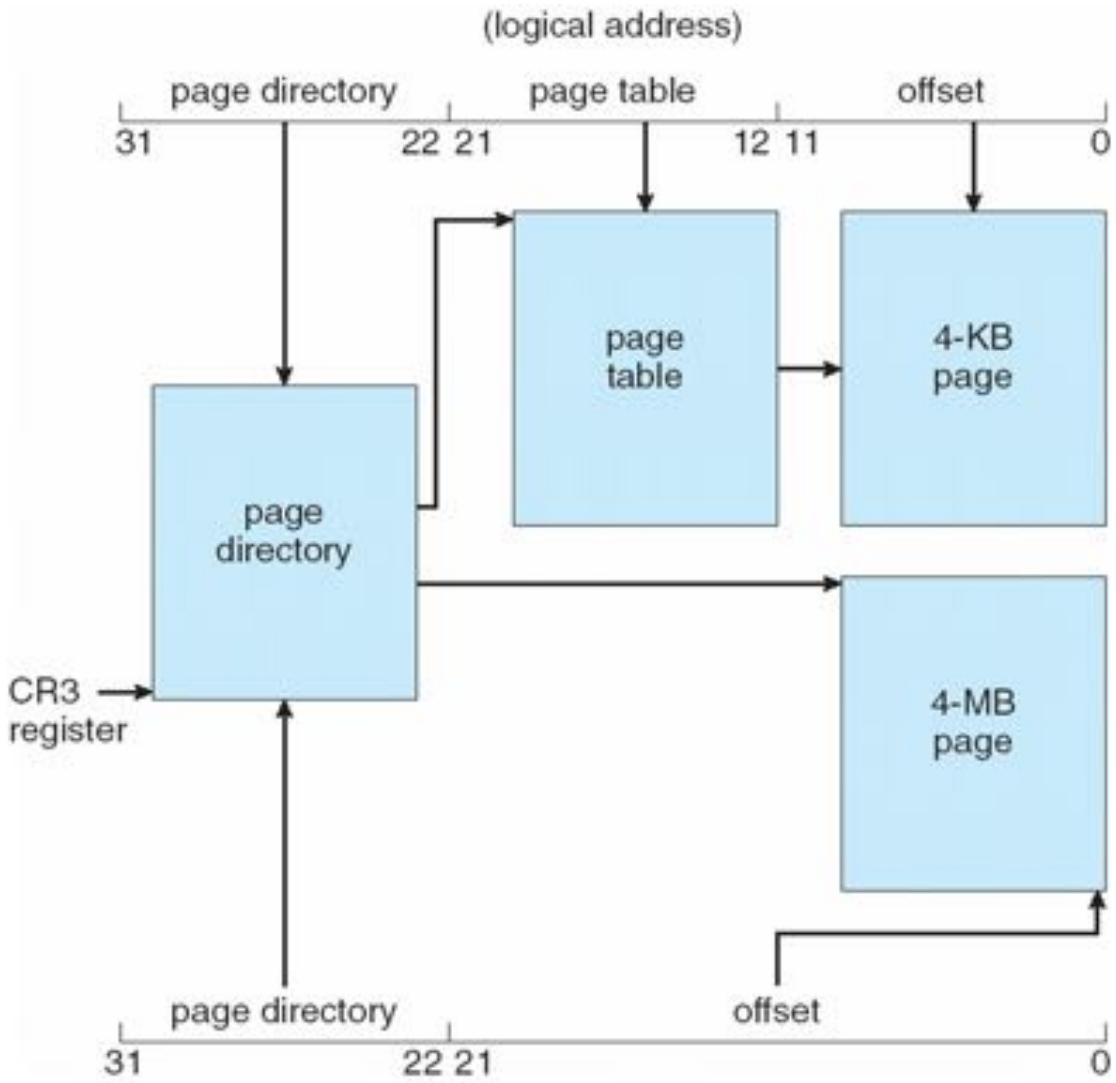


# *Intel IA-32 Segmentation*

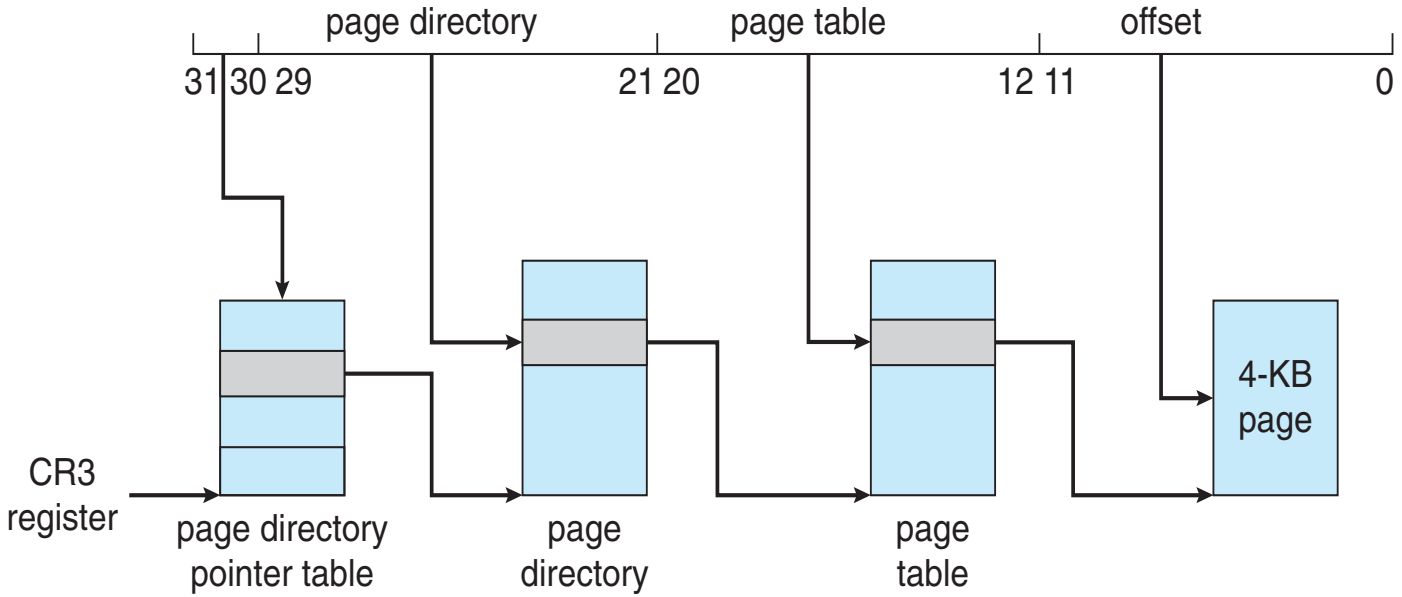


# *Intel IA-32 Paging Architecture*



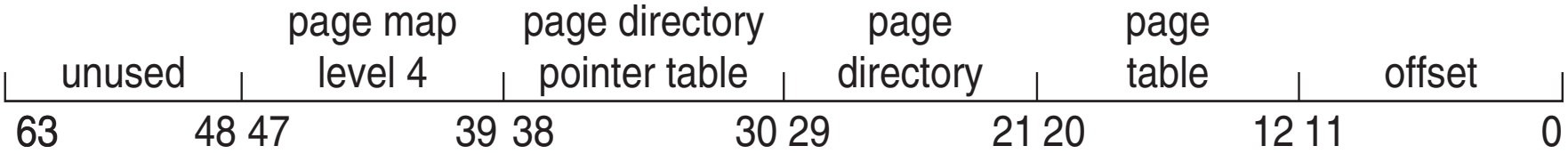
# *Intel IA-32 Page Address Extensions*

- 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
  - Increases address space to 36 bits (64GB physical memory)



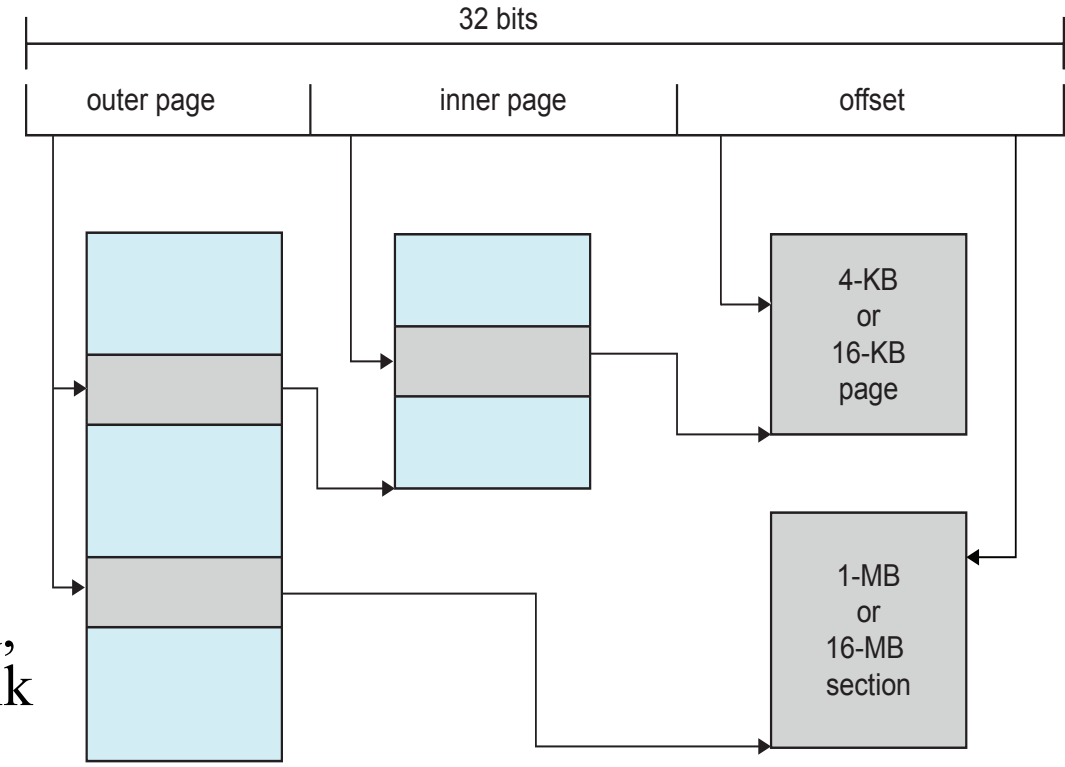
# *Intel x86-64*

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



# *Example: ARM Architecture*

- Dominant mobile platform chip
- 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



# *Next Class*

- Virtual memory