Week 6: Main Memory Management

Sherif Khattab

http://www.cs.pitt.edu/~skhattab/cs1550

(slides are from **Silberschatz, Galvin and Gagne ©2013)**

Administrivia

- Project 1 due on 2/21 @11:59pm
- Midterm on 2/22

Agenda

- Background
- Contiguous Memory Allocation
- Segmentation
- Paging
- Examples

Background

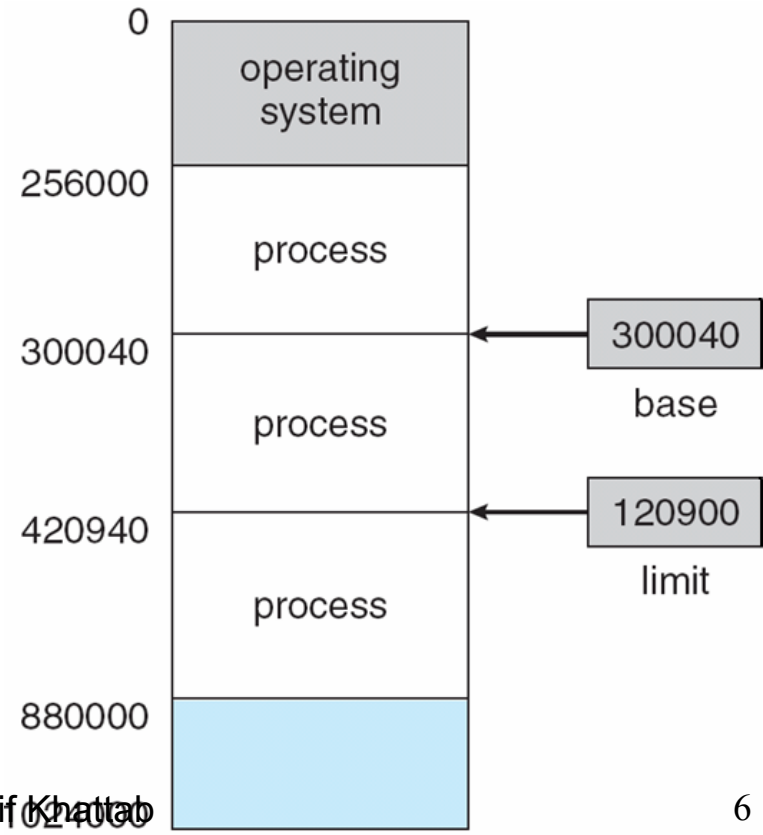
- Main memory and registers are the only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**

Sharing Main Memory

- Why do we want to share the main memory?
- Considerations:
  - Protection
    - A process cannot access other processes' memory nor the OS kernel memory
  - Abstraction
    - More programmer-friendly
    - Hide memory access details
    - Allow (virtual) process memory to exceed physical main memory
  - Performance
    - Affected by memory allocation

Protection: Base and Limit Registers

- A pair of **base** and **limit registers** define the 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
  - trap if error
- Changing these registers only in privileged (kernel) mode

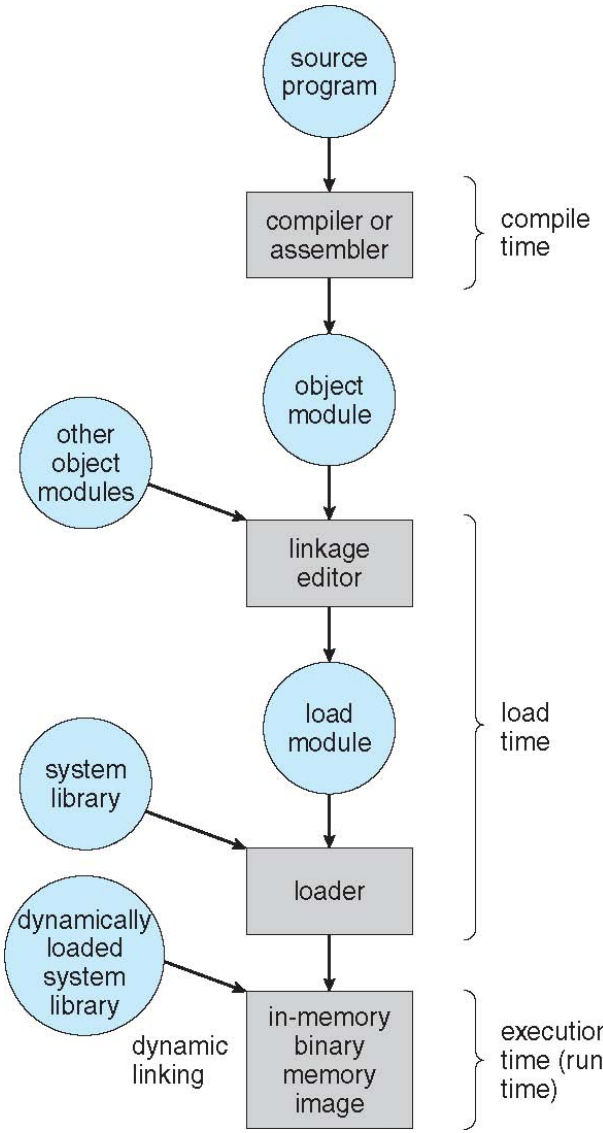


Abstraction

- Abstraction occurs on multiple levels:
  - symbolic addresses (e.g., int x, calling a method)
  - logical addresses
  - physical addresses

Symbolic Addresses

- Address binding of instructions and data 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 the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)



Dynamic Linking

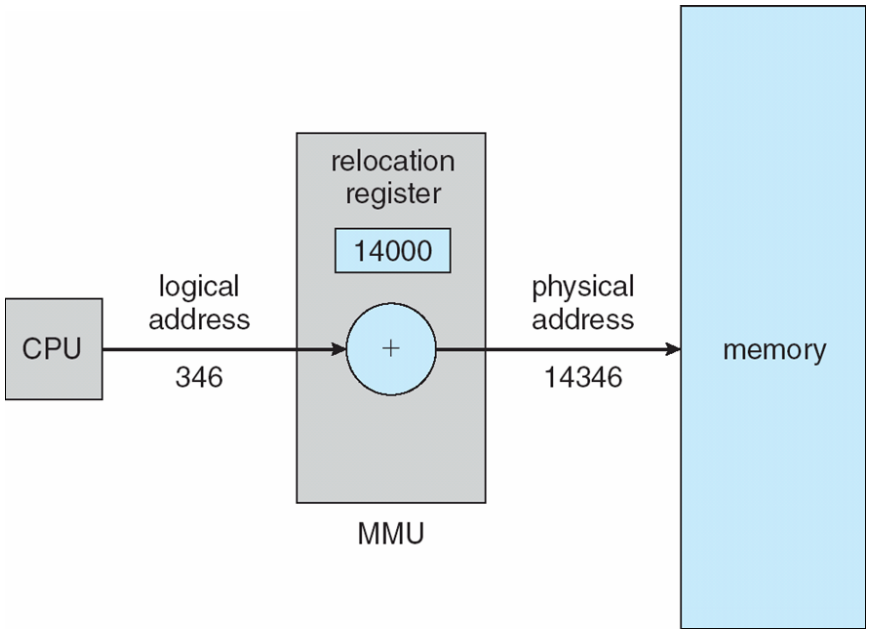
- Linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes' memory address
    - If not in address space, add to address space
- Dynamic linking is particularly useful for system libraries also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- When does logical address = physical address?

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- A simple scheme: the value in the relocation register (base register) is added to every address generated by a user process at the time it is sent to memory
  - No special support from the operating system is required

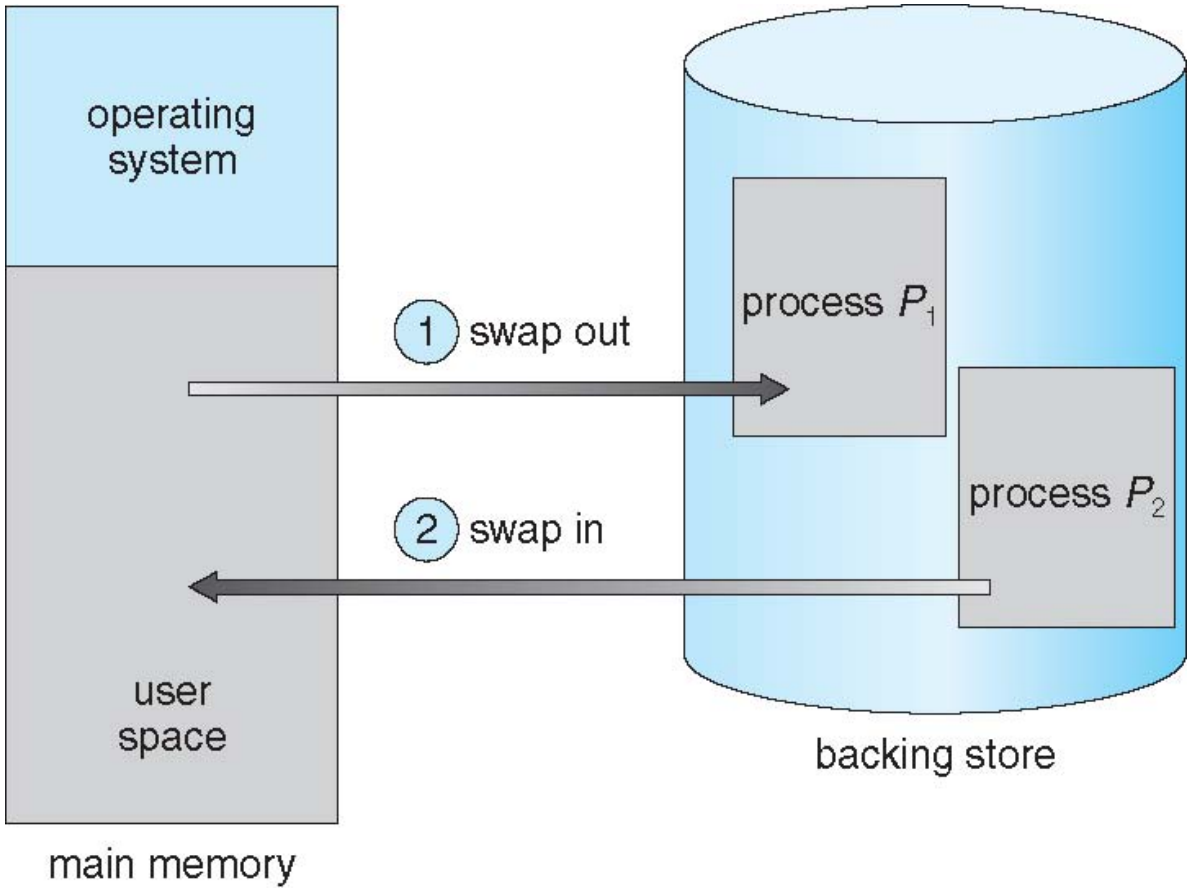


Performance

- Time per memory access
  - may need multiple memory accesses
- What if physical main memory is full?
  - swapping?

Swapping

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to and from process memory space => double buffering



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 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)
- Can reduce 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()`

Context Switch Time and Swapping (Cont.)

- 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
- Standard swapping not 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 on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data (e.g., code) 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 later

Memory Allocation

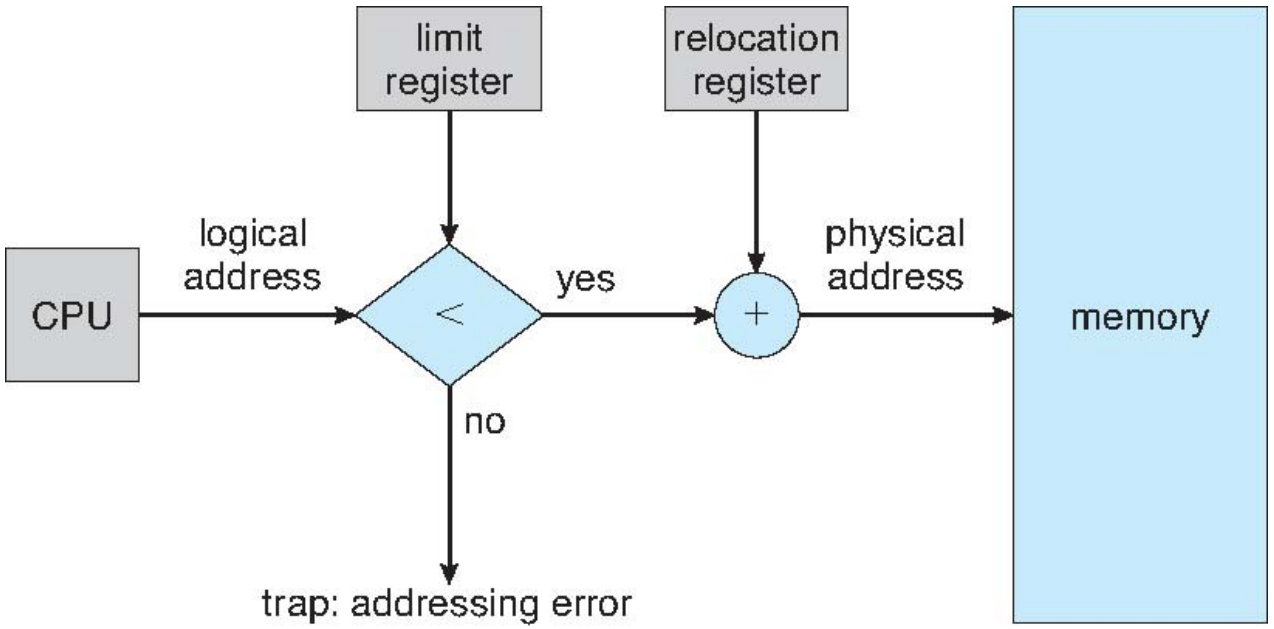
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Three main techniques
  - contiguous allocation
  - segmentation
  - paging

Contiguous Allocation

- Contiguous allocation is one early method
- Main memory usually partitioned 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 of memory
- Degree of multiprogramming limited by number of partitions
- Variable-partition sizes for efficiency (sized to a given process' needs)
- When a process arrives, it is allocated memory from a hole large enough to accommodate it

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size



Dynamic Storage-Allocation Problem

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

Fragmentation

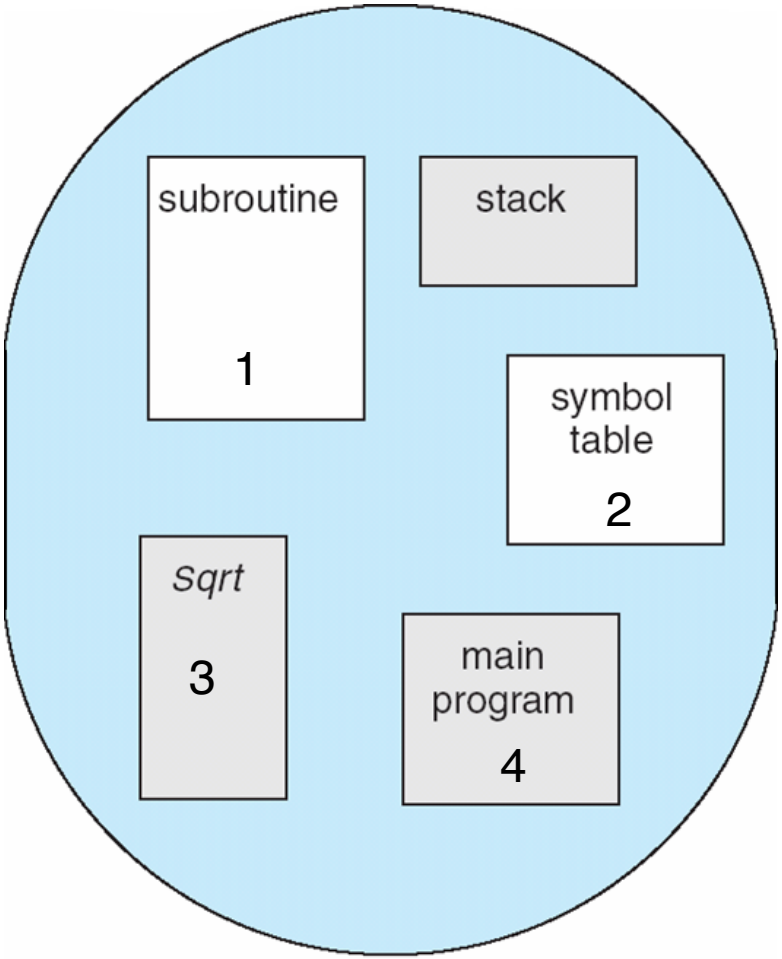
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

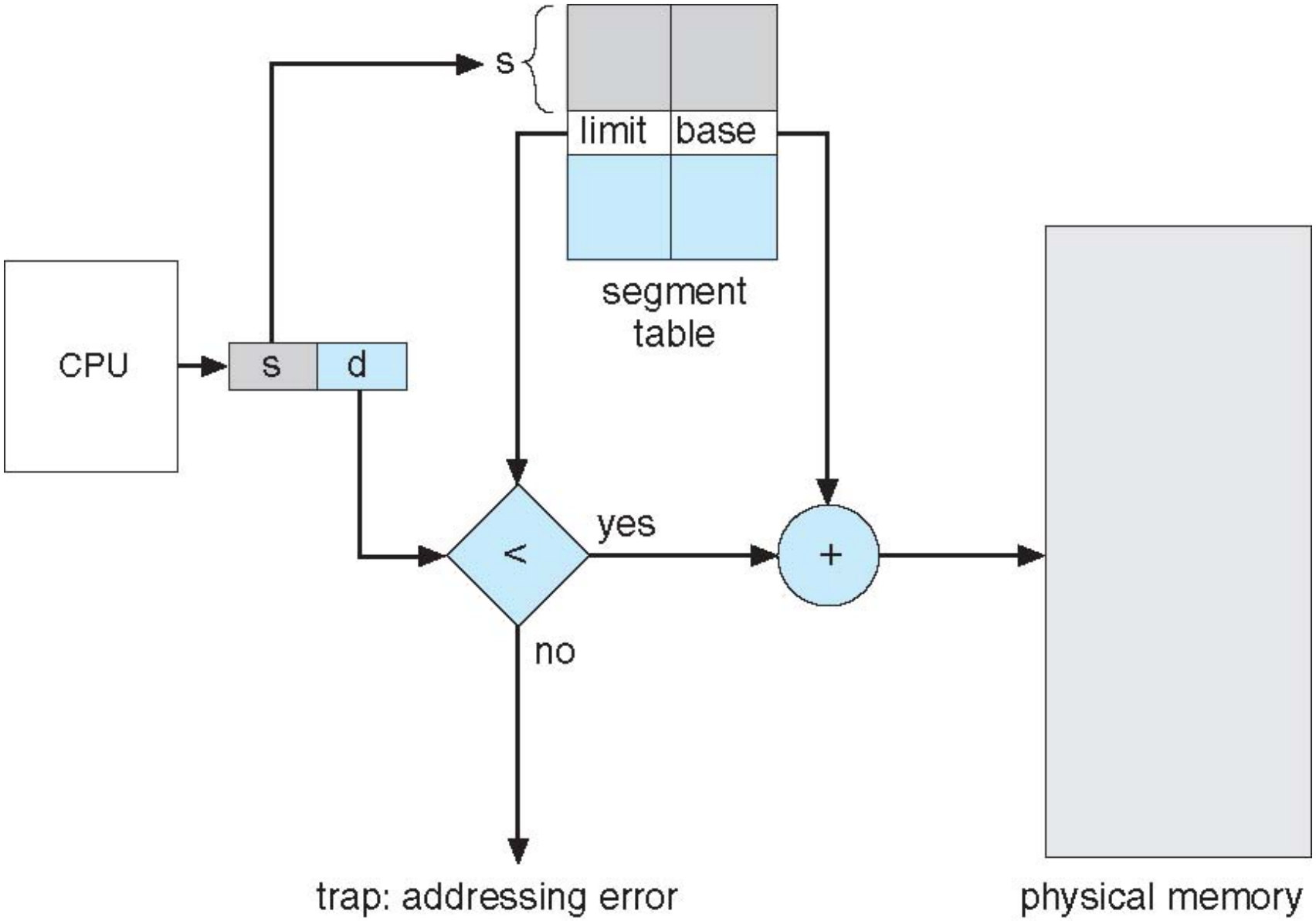
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem (again)
- Now consider that backing store has same fragmentation problems

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments



Segmentation Hardware

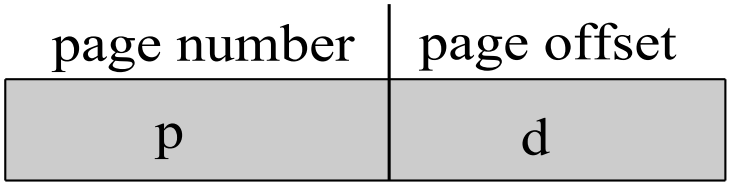


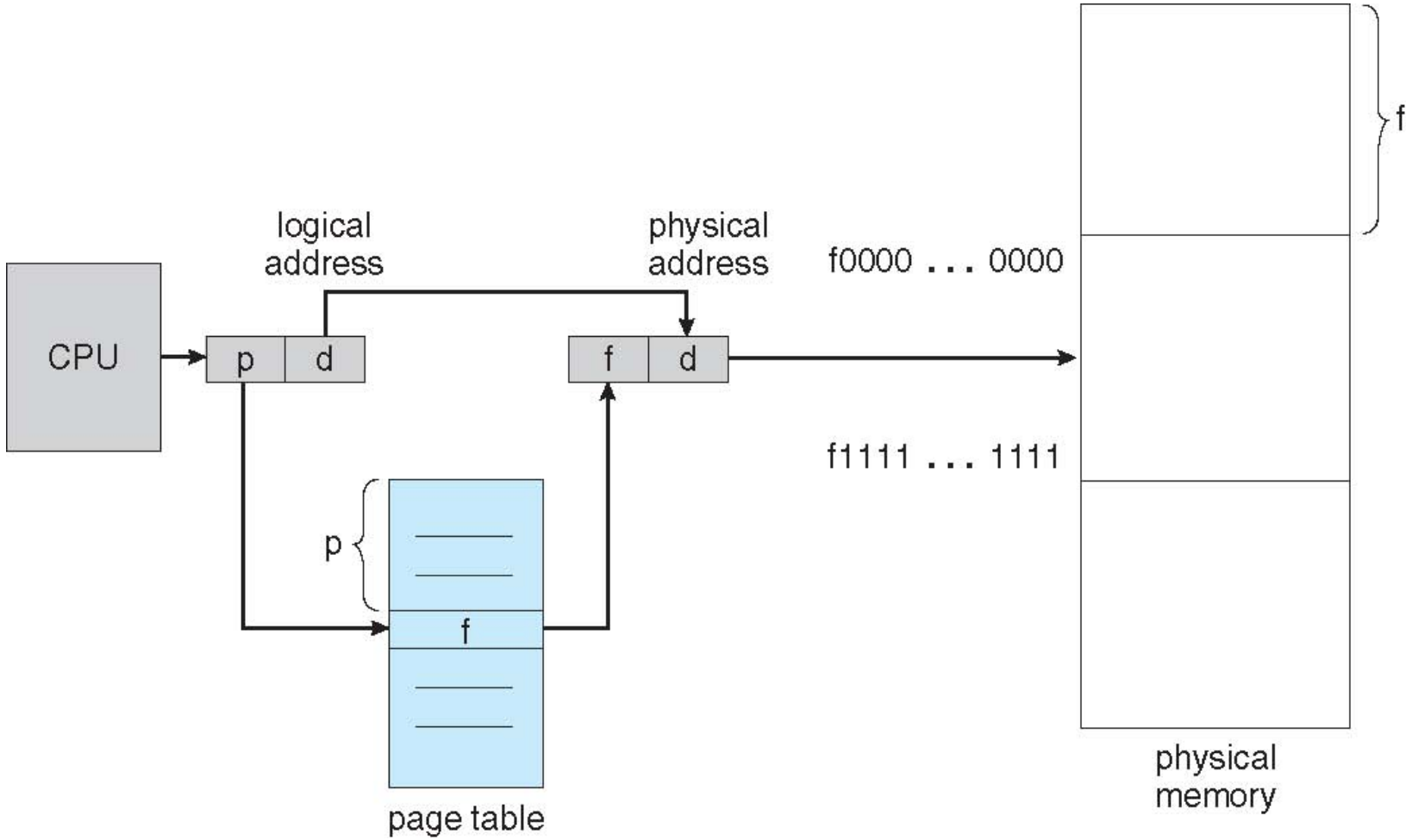
Paging

- Physical address space of a process can be noncontiguous
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- 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

Paging Hardware

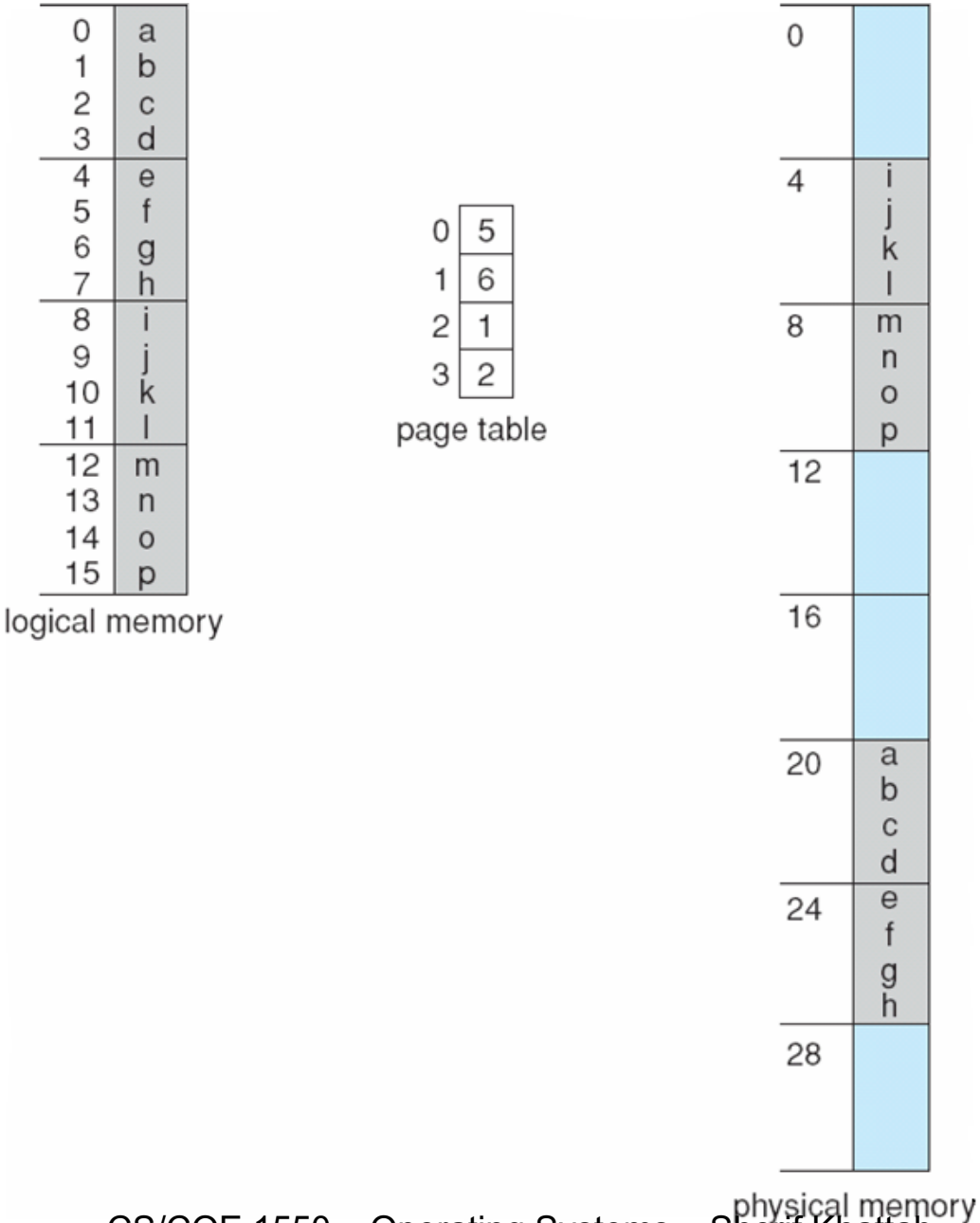
- For given logical address space 2^m and page size 2^n





Paging Example

n=2 (4-byte pages), *m*=4, 32-byte memory



Internal 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

How to store the Page Table?

- Page table is kept in main memory
- Alternatives:
  - One contiguous piece
  - Page the page table itself!
- Issues:
  - What about unused entries?
    - not all pages are used/needed by all processes
  - Effect on performance
  - how to implement protection?
  - How to implement sharing?
  - How big can the page table be?

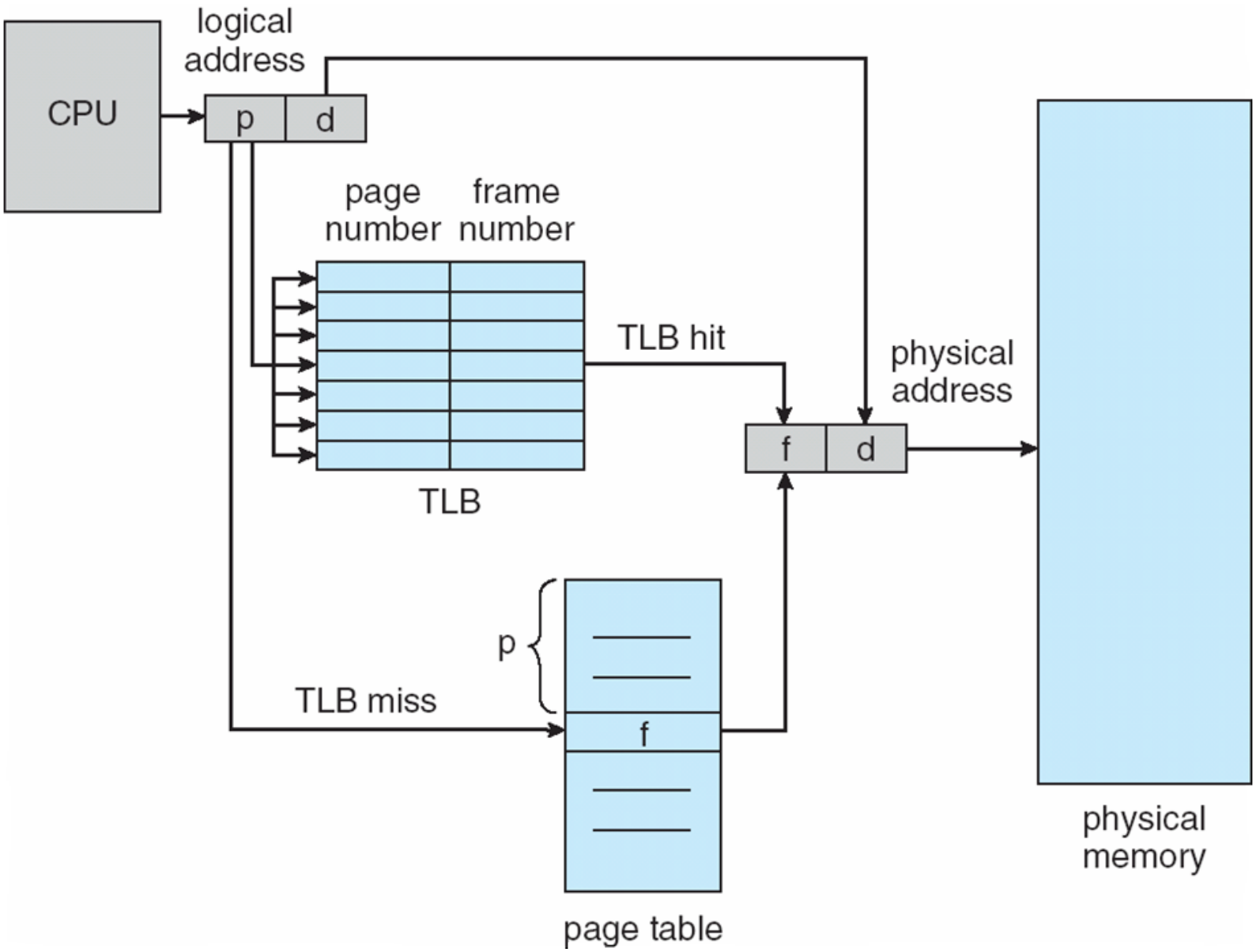
Contiguous Storage of Page Table

- **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
- The two memory accesses problem can be solved by the **translation look-aside buffers** (**TLBs**)

Translation Look-aside Buffers (TLBs)

- fast-lookup hardware cache
- uses **associative memory**
- Lookup implemented within CPU pipeline
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
  - data TLB and instruction TLB
- 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 (kernel?)

Paging Hardware With TLB



Effective Access Time

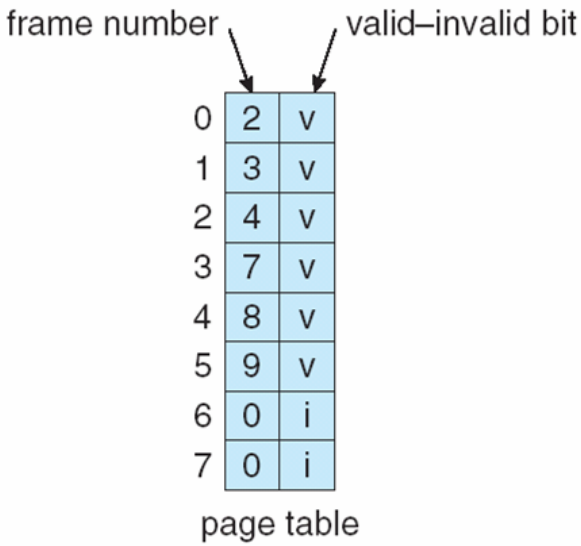
- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
  - Hidden within the pipeline
- Hit ratio (α): percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time** (**EAT**)

  EAT = $(1 + \varepsilon)\ \alpha + (2 + \varepsilon)(1 - \alpha)$ main memory accesses

  $= 2 + \varepsilon - \alpha$ main memory accesses
- Consider α = 80%, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%:
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

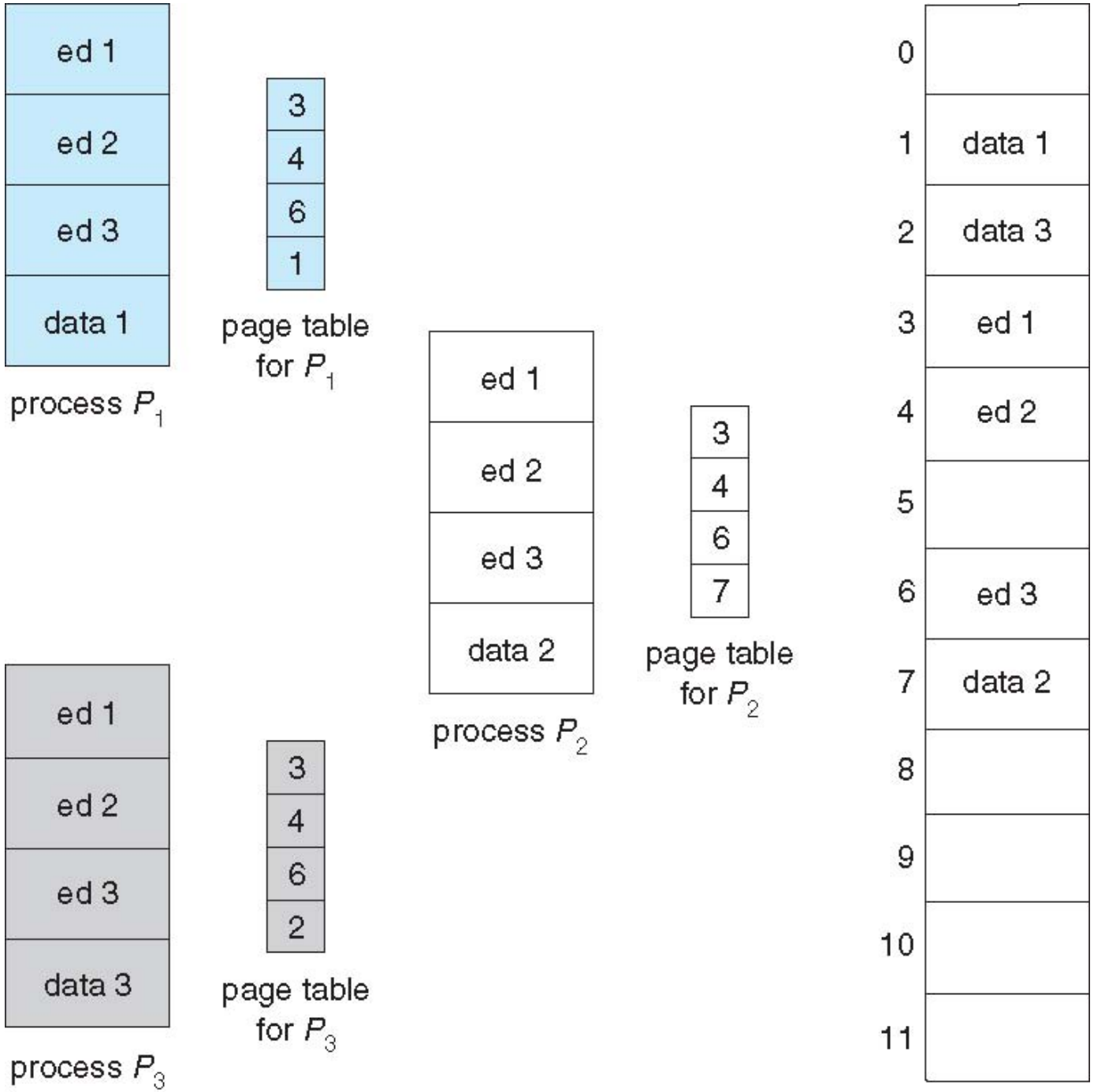
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - Or **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel



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

Shared Pages Example

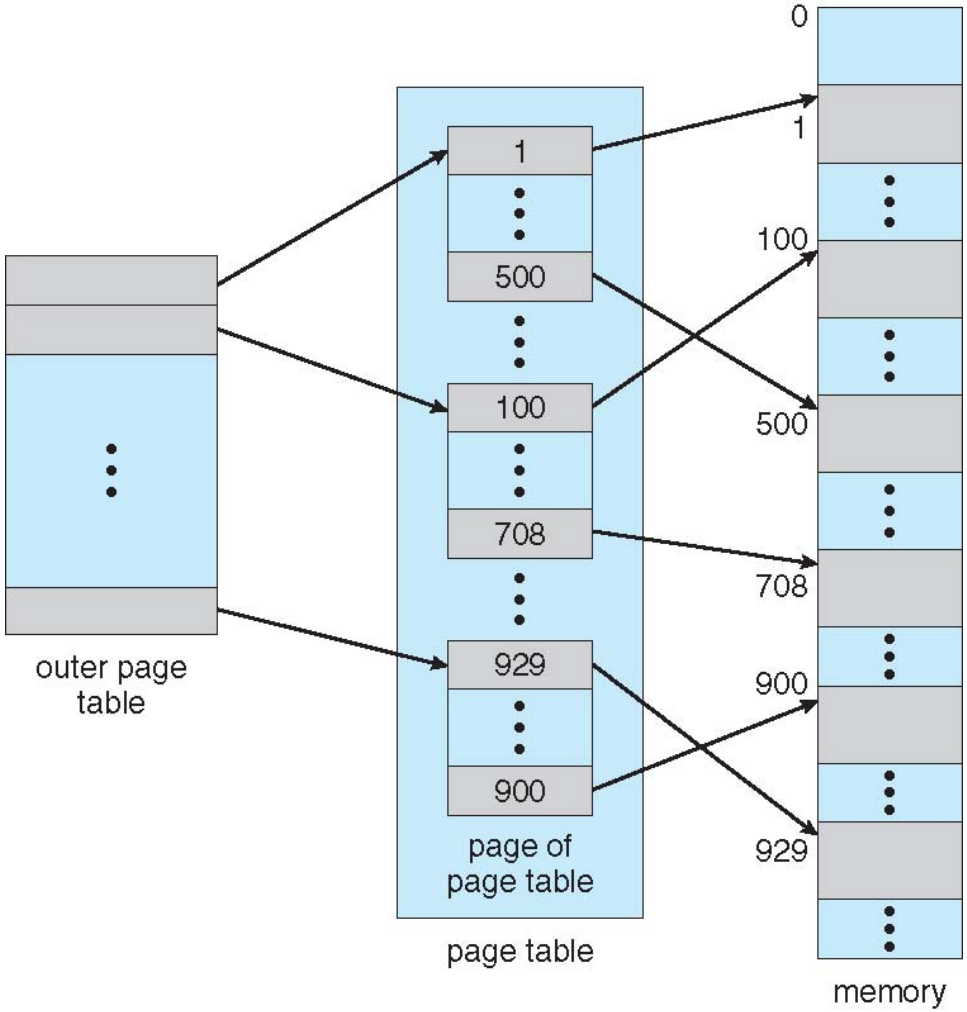


Size of the Page Table

- 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
    - Don't want to allocate that contiguously in main memory
- Solution:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

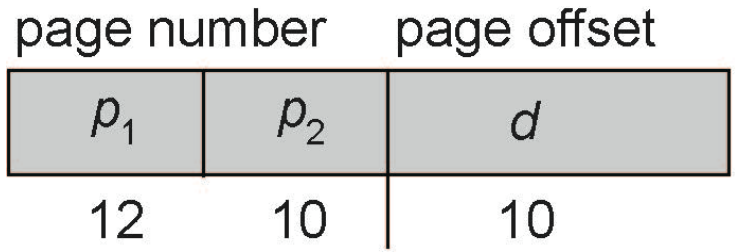
Hierarchical Page Tables

- Break up the page table into multiple smaller page tables
  - Two-level page table
  - We page the page table



Two-Level Paging Example

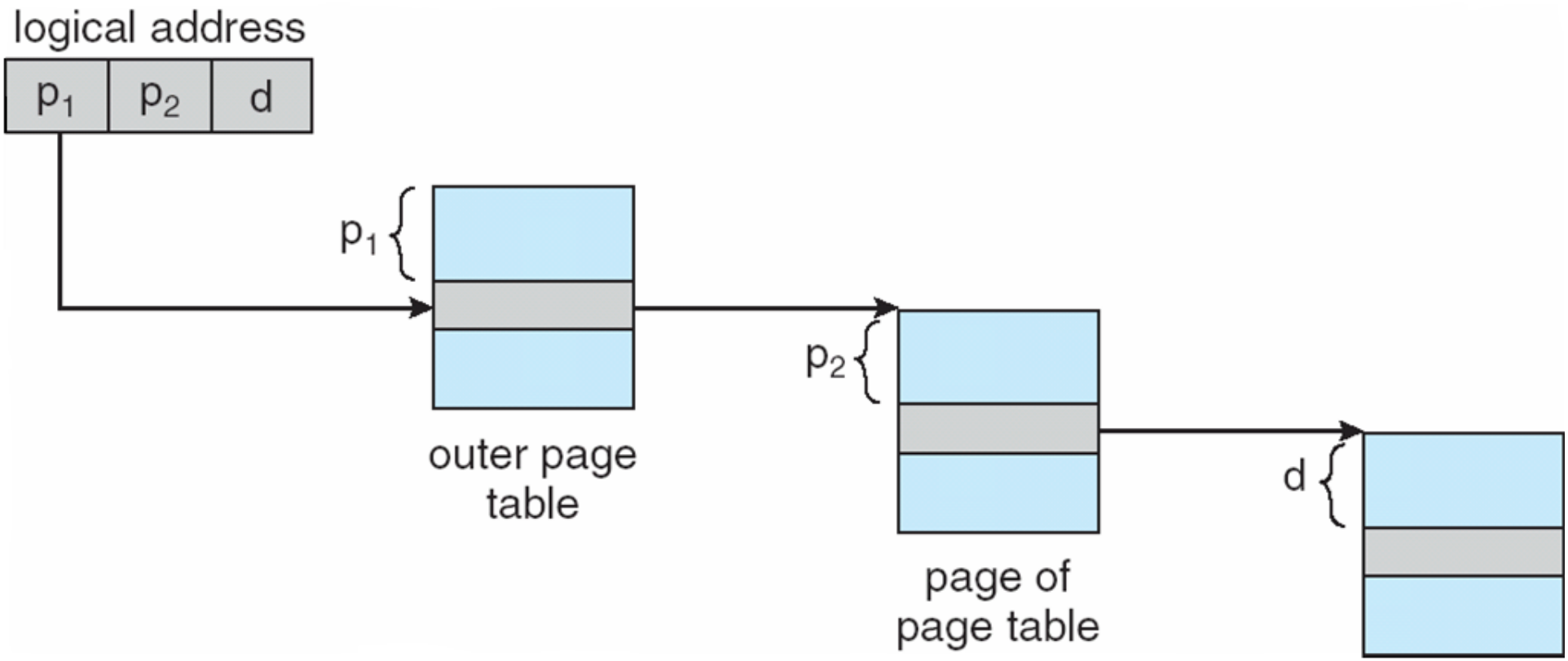
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset (why 10 bits?)



- 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**

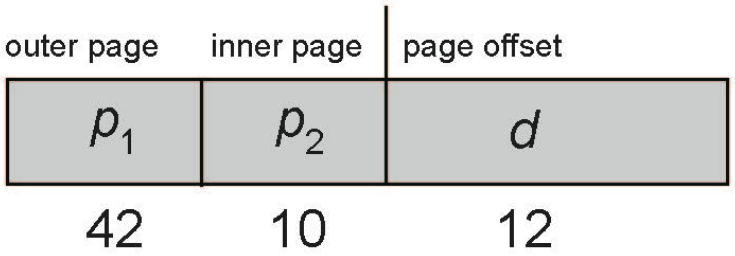
Address-Translation Scheme

- three memory-accesses at worst



64-bit Logical Address Space

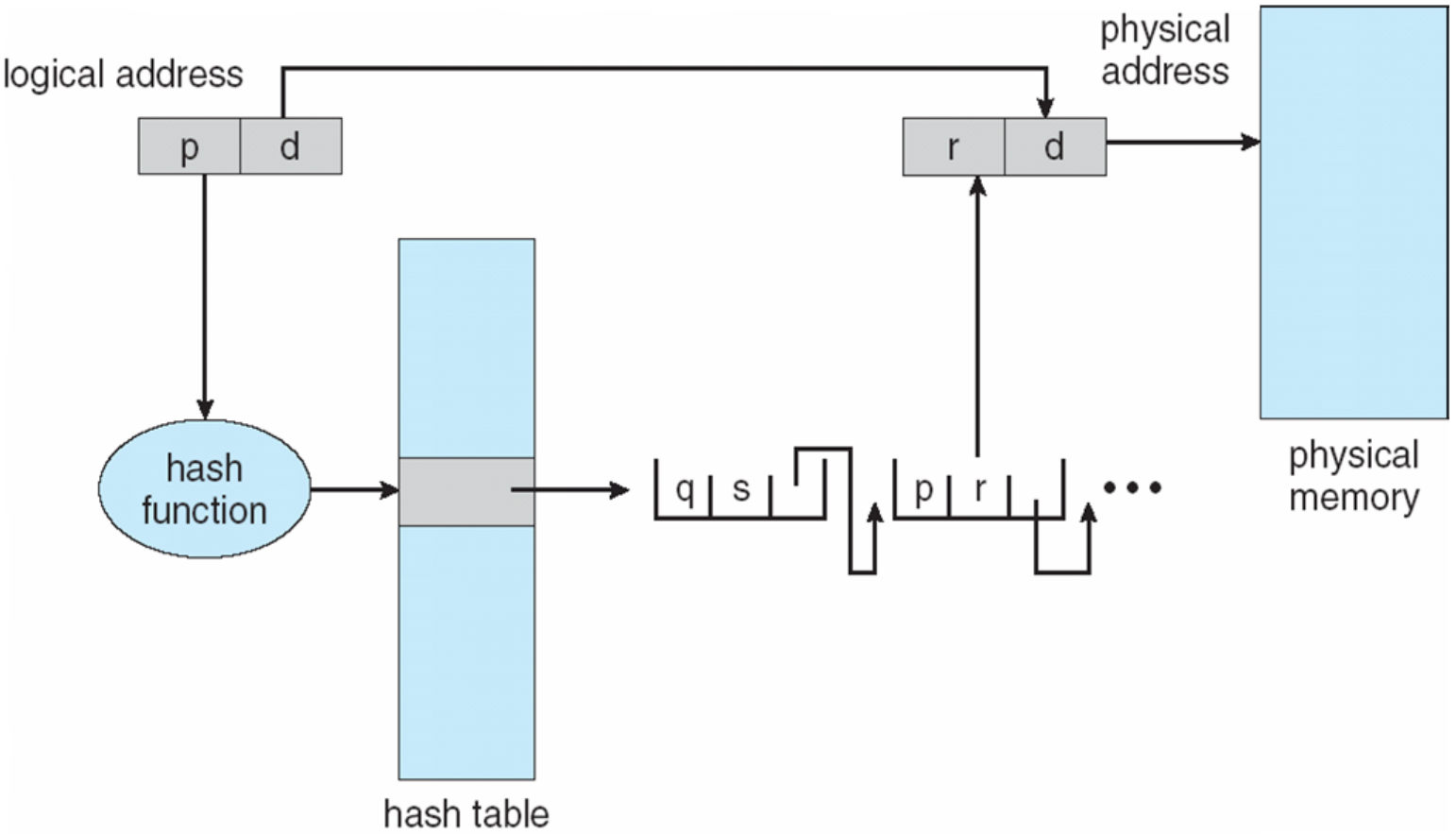
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And 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 (open addressing)

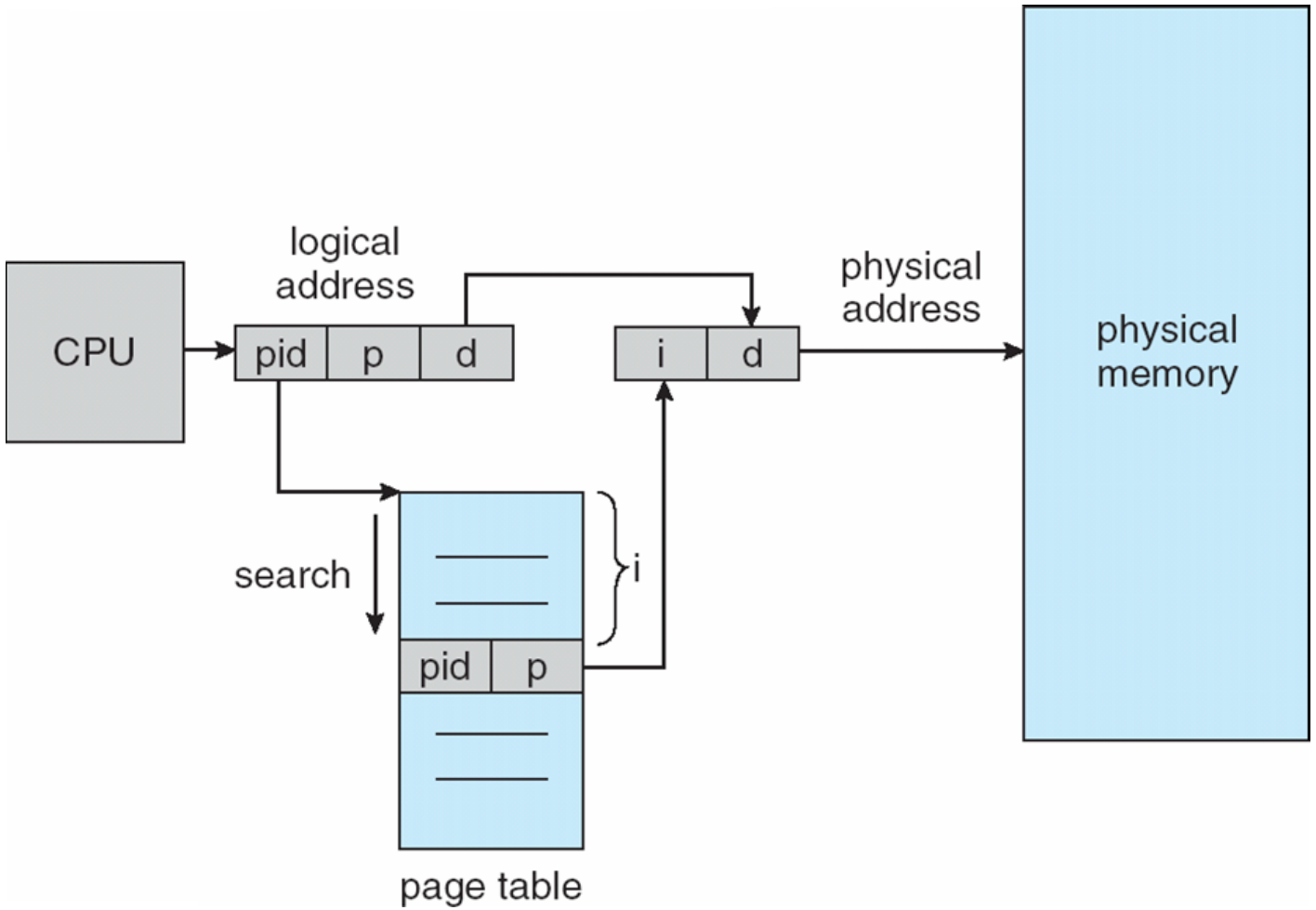


Hashed Page Table with clustering

- 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)

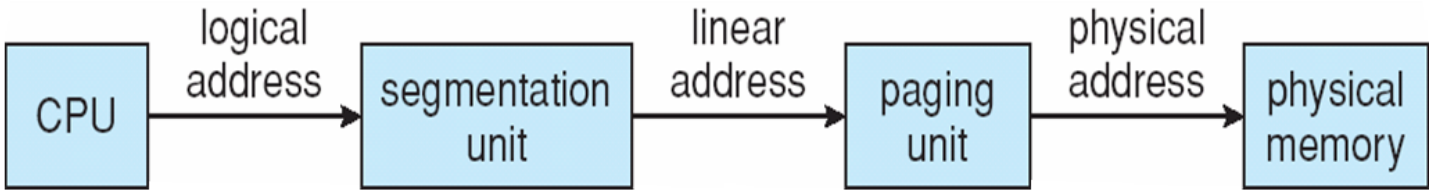
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages; need to search.



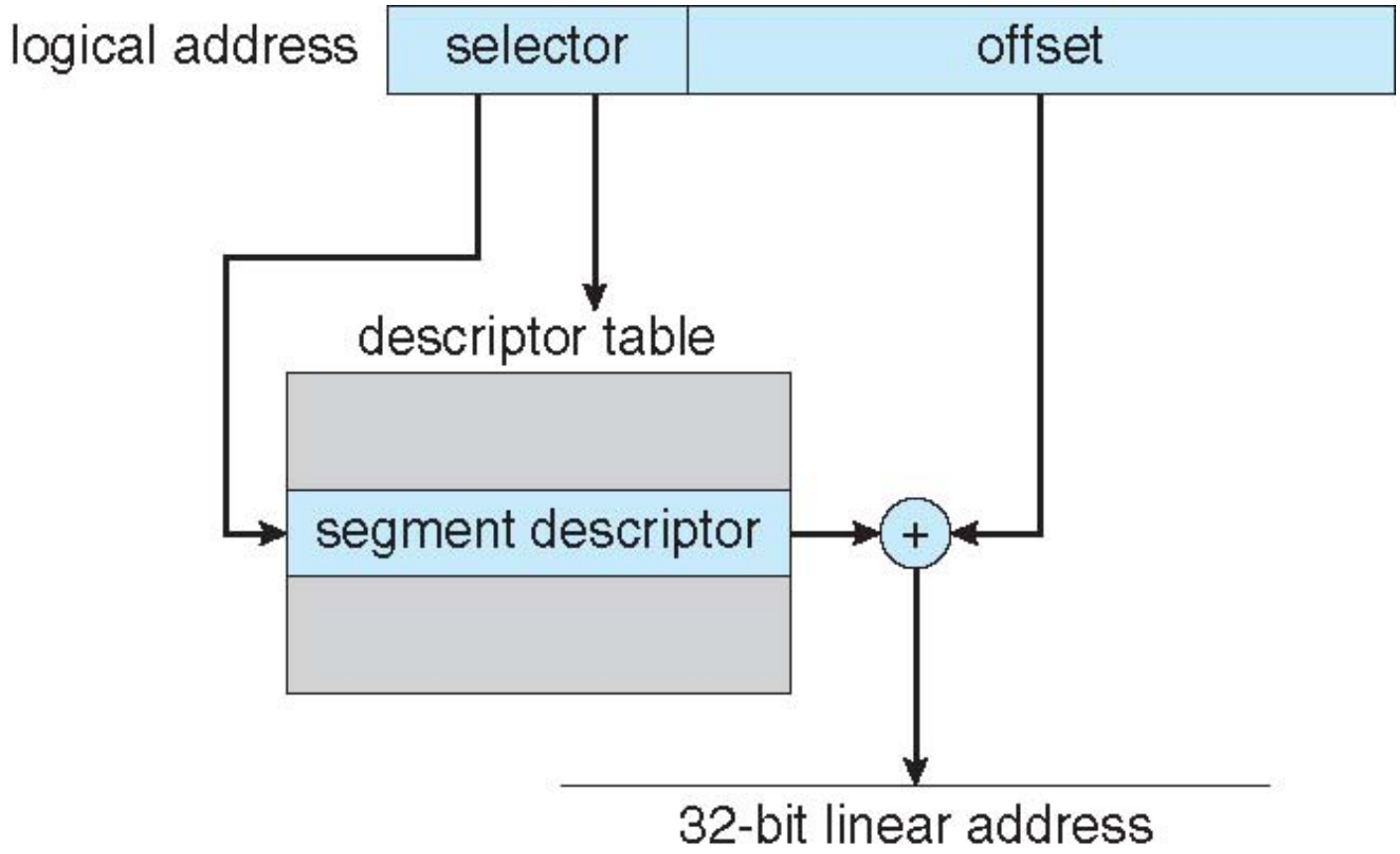
Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16K segments per process
  - Logical address space divided into two partitions
    - First partition of up to 8K 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**))
    - 8-byte segment entry (base, limit, protection, etc.)
  - Pages sizes can be 4 KB or 4 MB

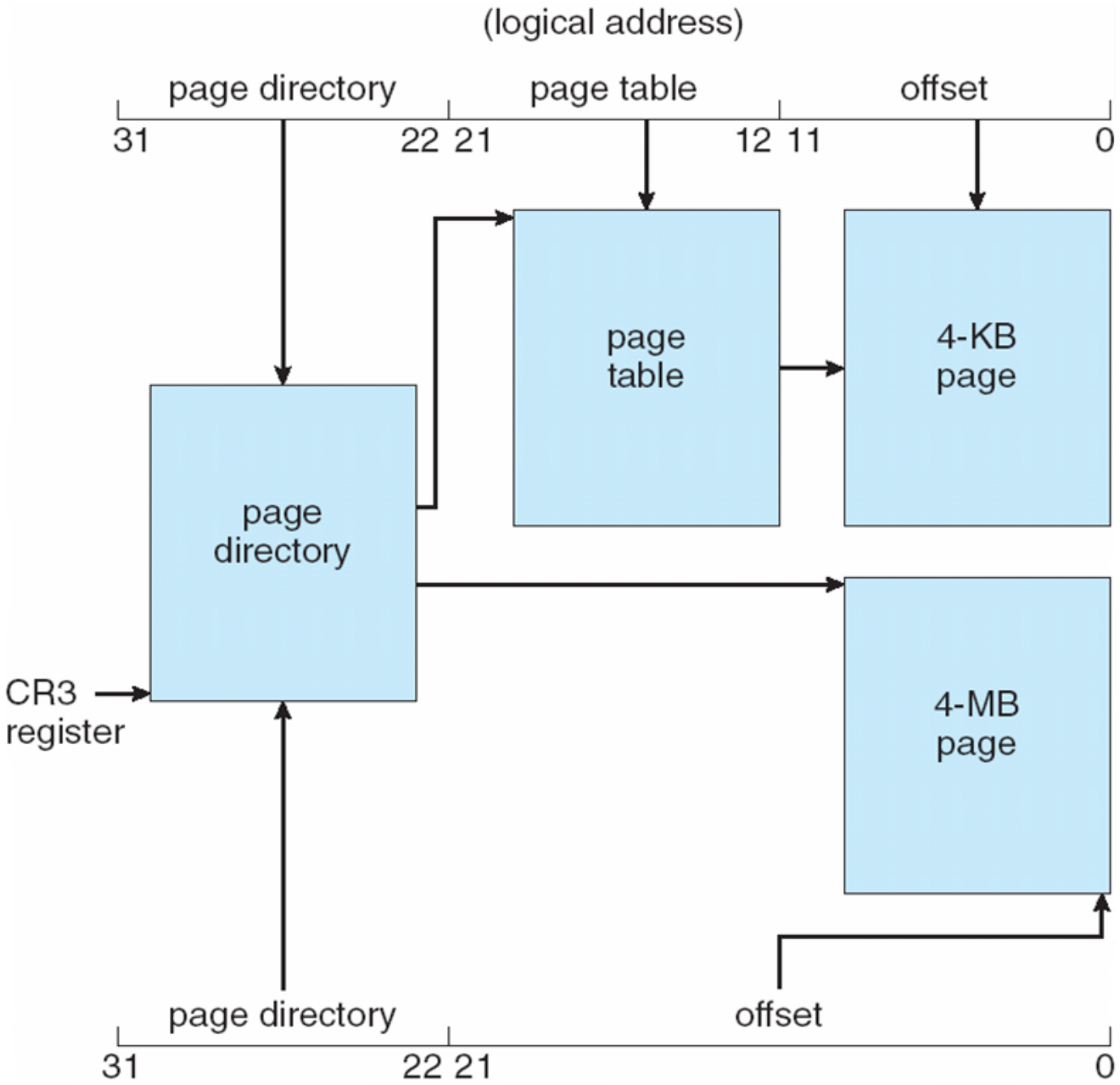


Intel IA-32 Segmentation

- selector: 16 bits
  - segment number (13 bits)
  - global or local (1 bit)
  - protection bits
- offset: 32 bits

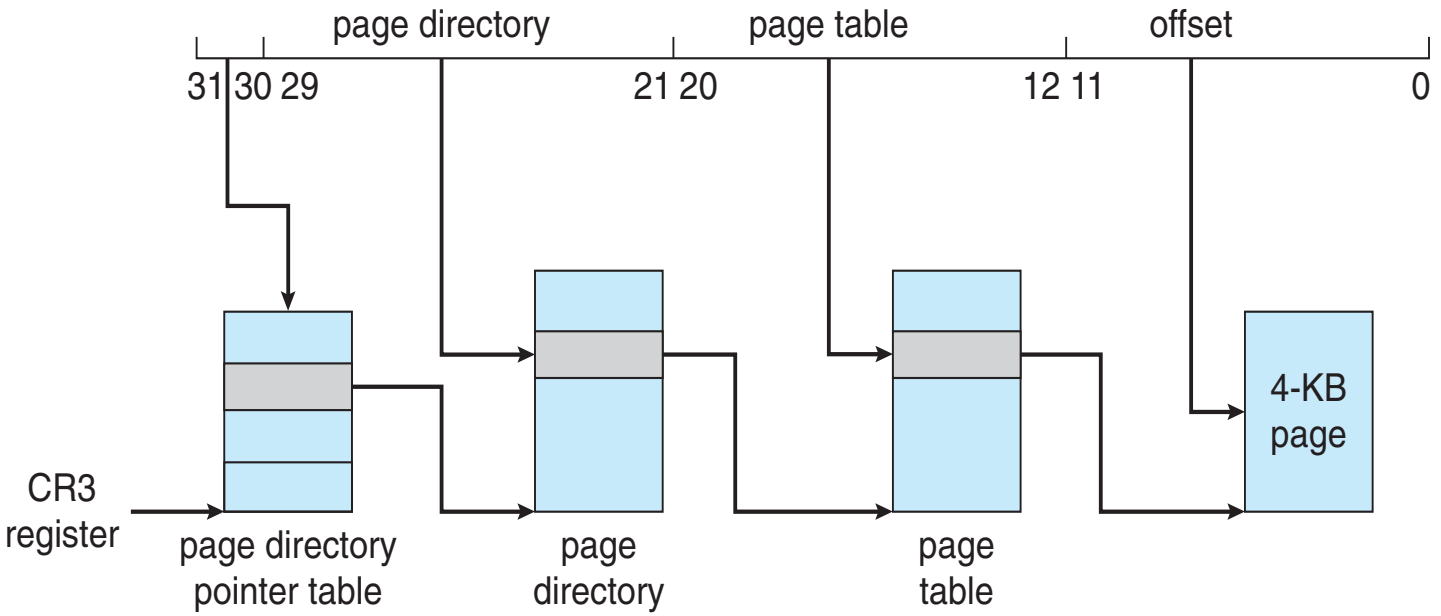


Intel IA-32 Paging Architecture



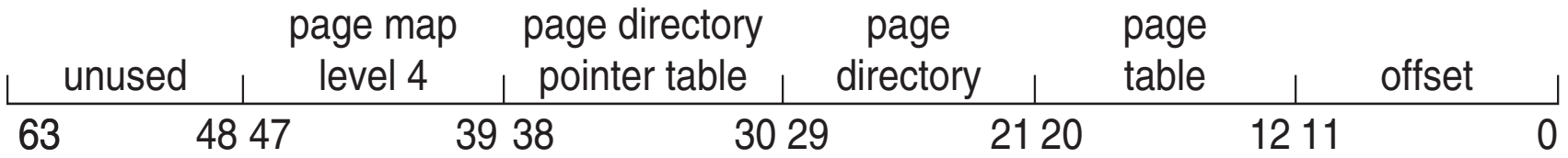
Intel IA-32 Page Address Extensions

- 32-bit apps allowed 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
  - 24-bit base + 12-bit offset



Intel x86-64

- Intel followed AMD this time.
- 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 (Apple iOS and Google Android devices for example)
  - ARM designs and licenses to manufacturers
- Modern, energy efficient, 32-bit CPU
- One-level paging
  - 1 MB and 16 MB pages (termed sections)
- two-level paging
  - 4 KB and 16 KB pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
- First outer TLBs are checked, on miss inner TLB is checked, and on miss page table walk performed in hardware