

Lecture 09: Main Memory



Slides Sources

- These slides are originated by the slides from the book
 - <https://www.os-book.com/OS10/slide-dir/index.html>
- Slides are adapted and extended by Sashko Ristov
 - Chapter 9

Semester Outline

- Introduction
- Processes
- Threads
- Interprocess Communication (IPC)
- Process Synchronization
- Scheduling
- Deadlocks
- **Memory Management**
- Virtual Memory
- Multicomputers and Multiprocessors
- I/O & Storage
- Virtualization Concepts
- SW/HW Virtualization

Outline

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

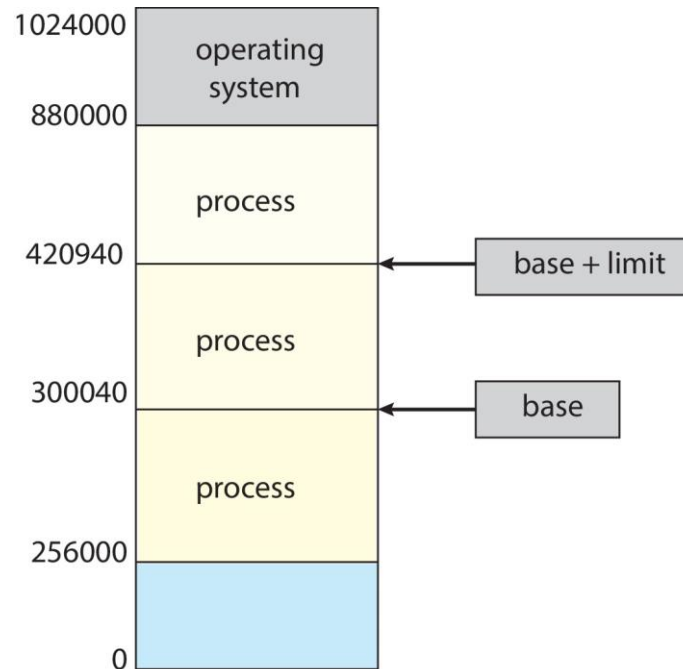
Background

Background

- **Program** must be **brought** (from disk) **into memory** and placed within a **process** for it to be **run**
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
 - addresses + read requests (returns a value), or
 - address + data and write requests (writes a value)
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

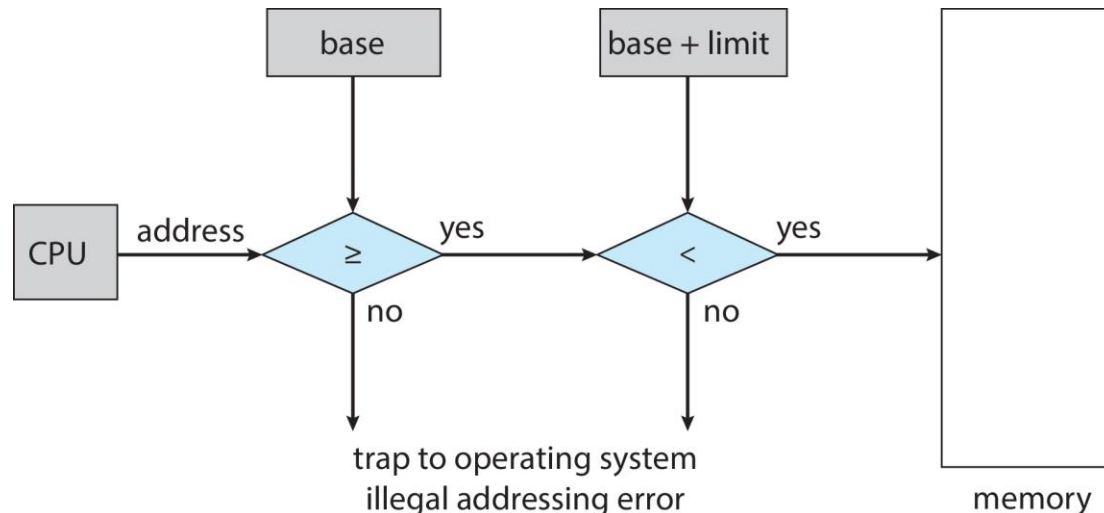
Protection

- Need to ensure that a **process** can access only those addresses in its **address space**.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



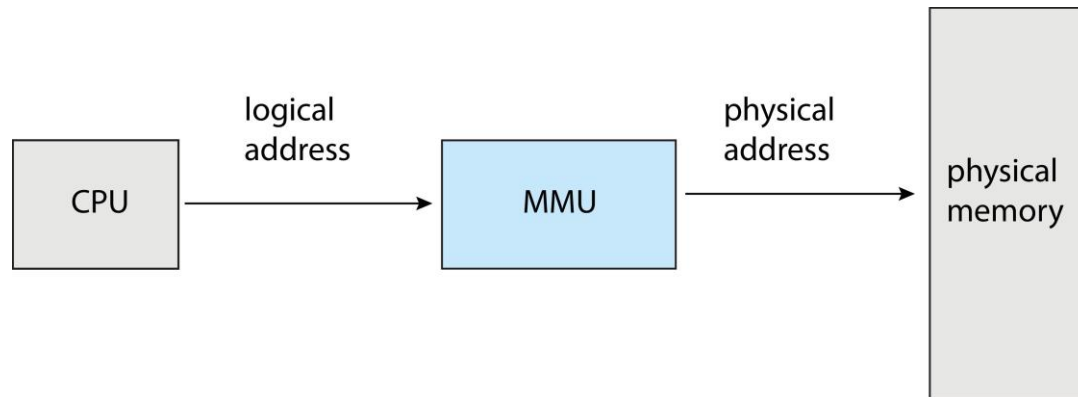
- the instructions to loading the base and limit registers are privileged

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** – 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

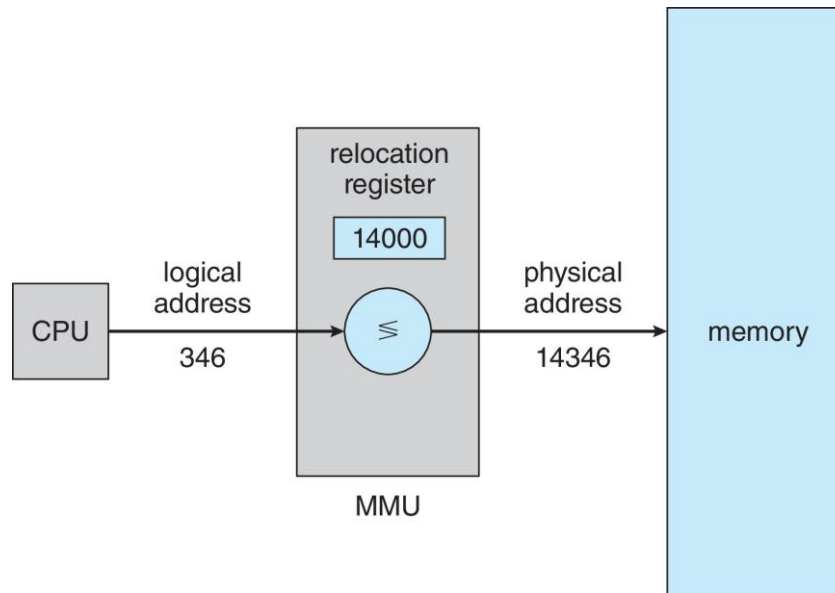
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



Memory-Management Unit (Cont.)

- Consider simple scheme, which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the **relocation register** is **added to every address** generated by a user process at the time it is sent to memory



Contiguous Memory Allocation

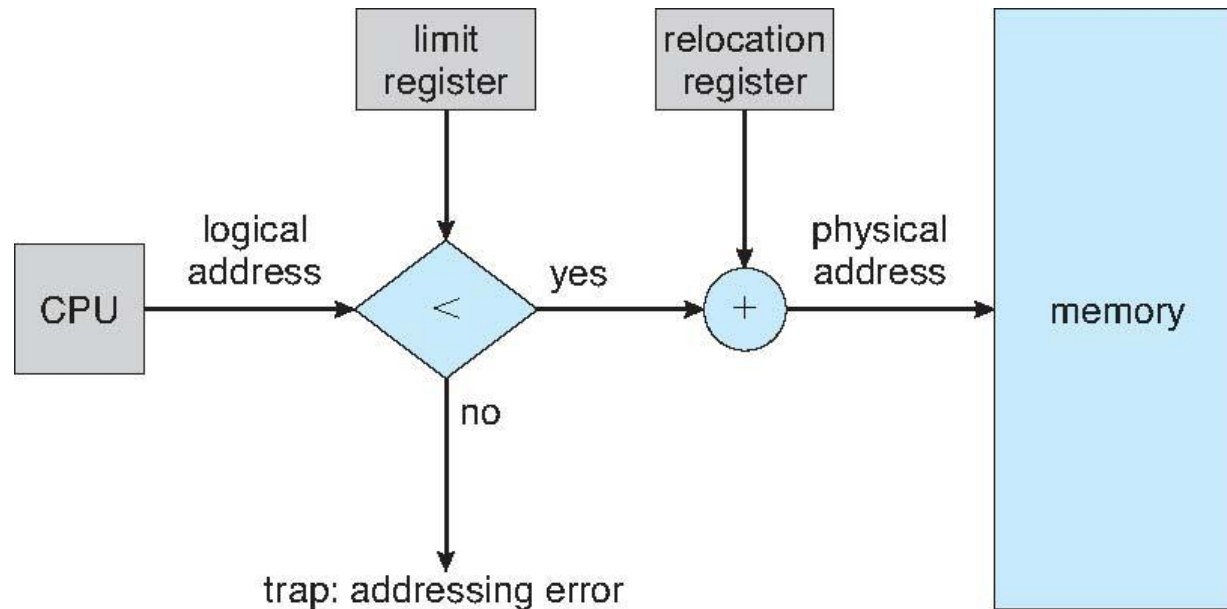
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate 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 of memory

Contiguous Allocation (Cont.)

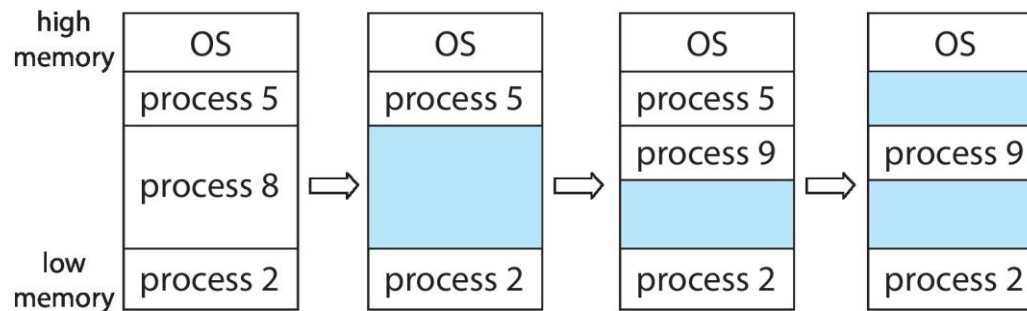
- **Relocation registers** used to **protect** user **processes** from each other, and from changing operating-system code and data
 - Base register contains value of the smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Variable Partition

- 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 from 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** – 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
 - A high overhead to keep track of small holes
 - Solution
 - ▶ break the memory into fixed-sized blocks
 - ▶ Allocate memory in units based on block size
- First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation ($0.5N/1.5N$)
 - $1/3$ may be unusable -> known as **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
 - ▶ Latch job in memory while it is involved in I/O
 - ▶ Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

Q/A



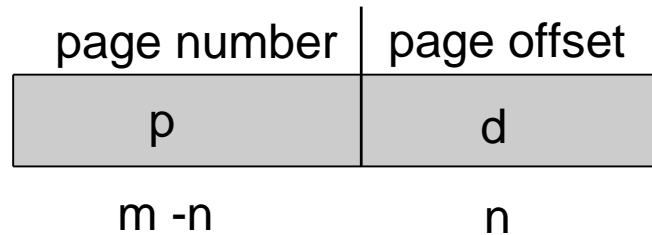
Paging

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- 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

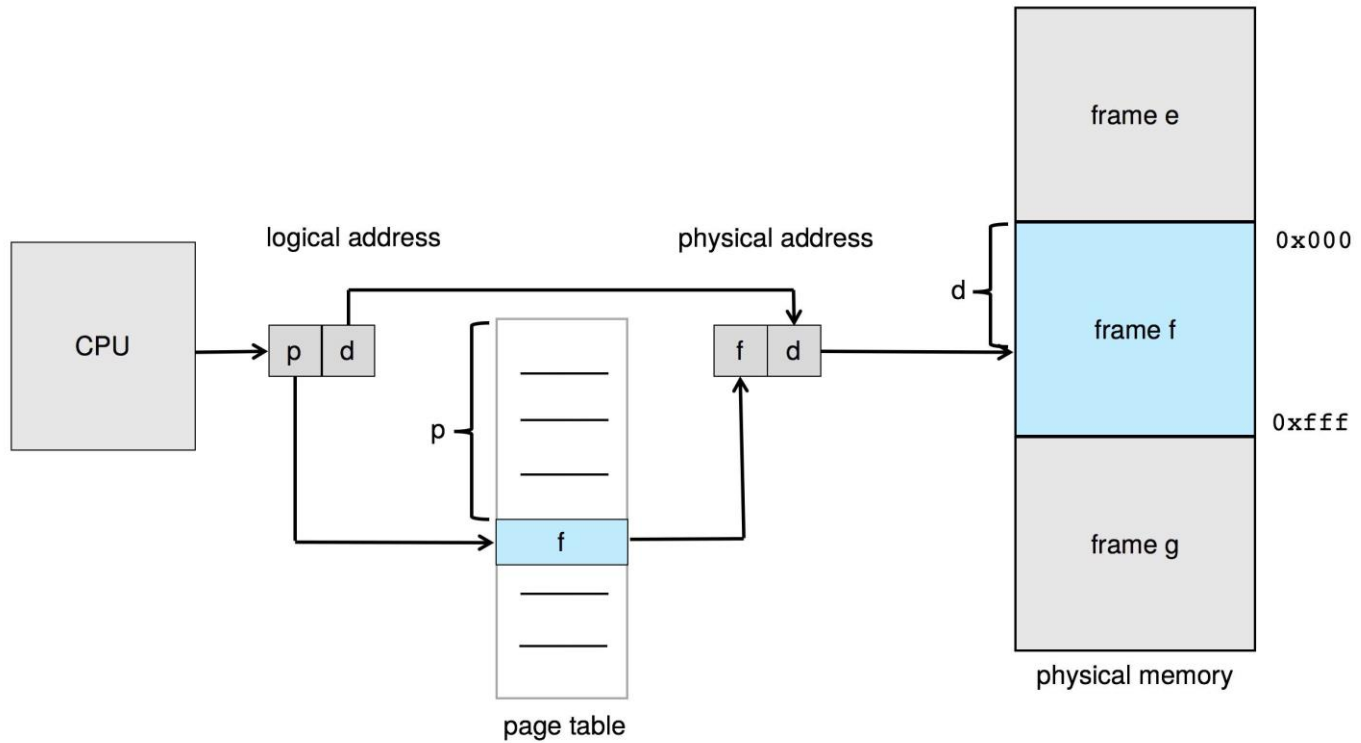
Address Translation Scheme

- 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

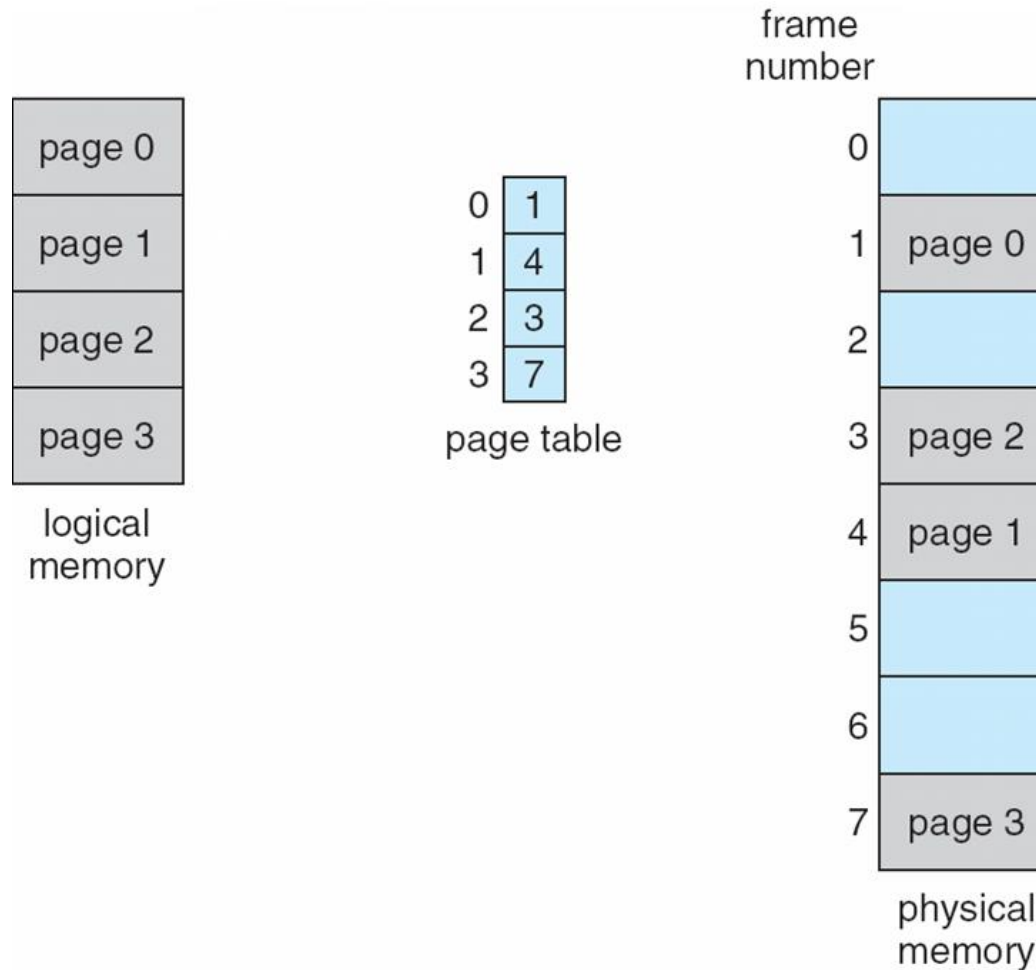


- For given logical address space 2^m and page size 2^n

Paging Hardware

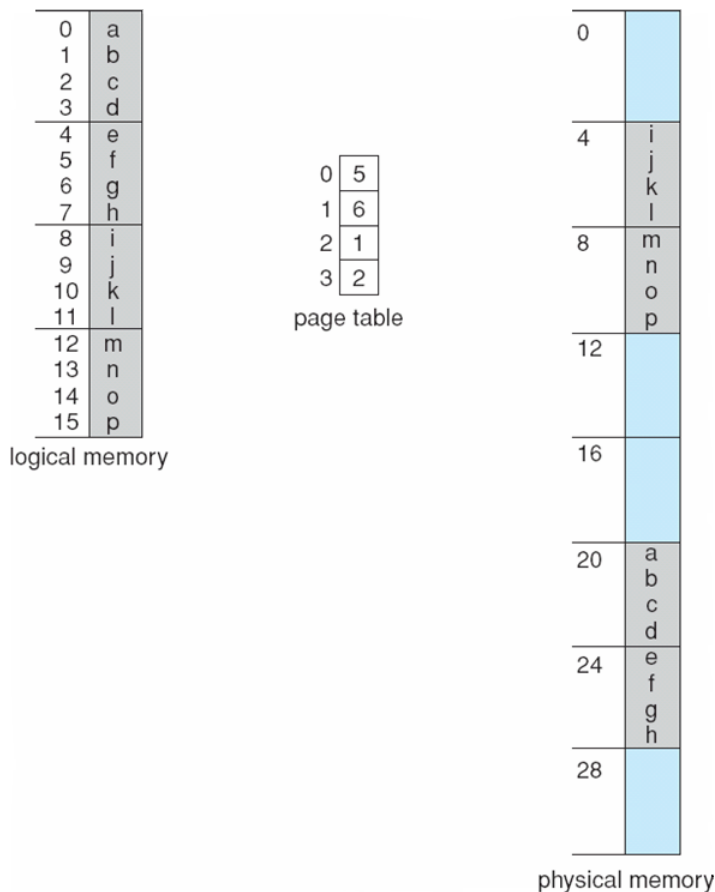


Paging Model of Logical and Physical Memory



Paging Example

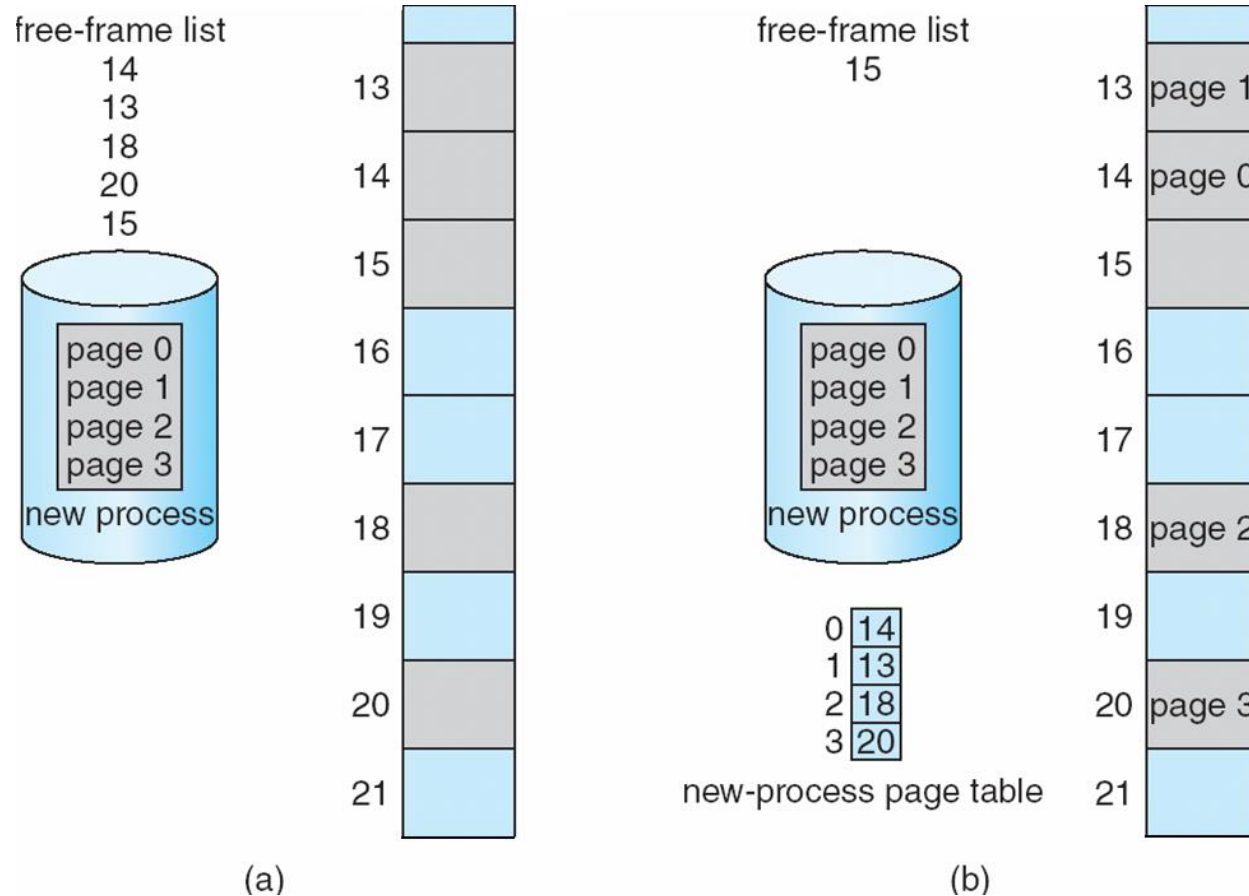
- Logical address: $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages, from page 0 to page 7)



Paging -- 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

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
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs)** (also called **associative memory**).

Translation Look-Aside Buffer

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

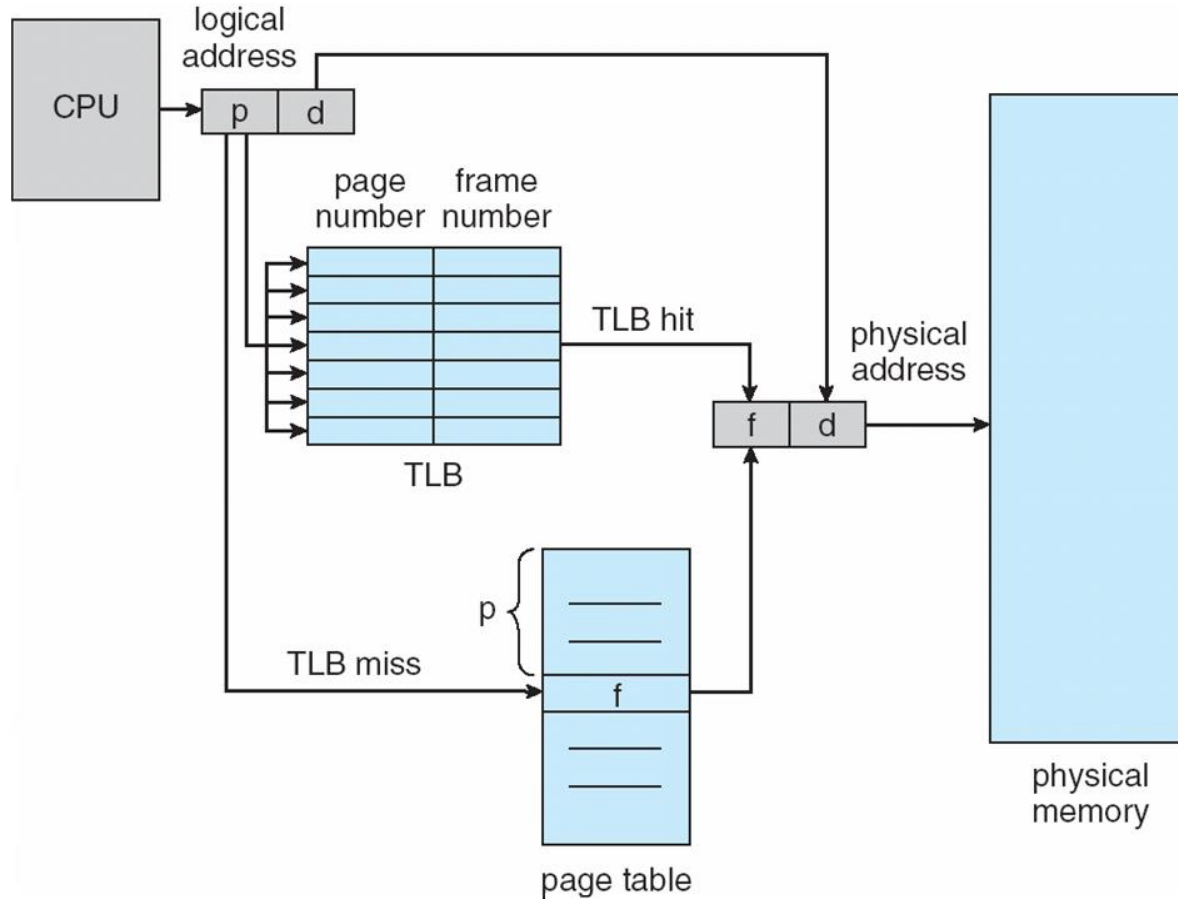
Hardware

- 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

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If we find the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time (EAT)**
$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

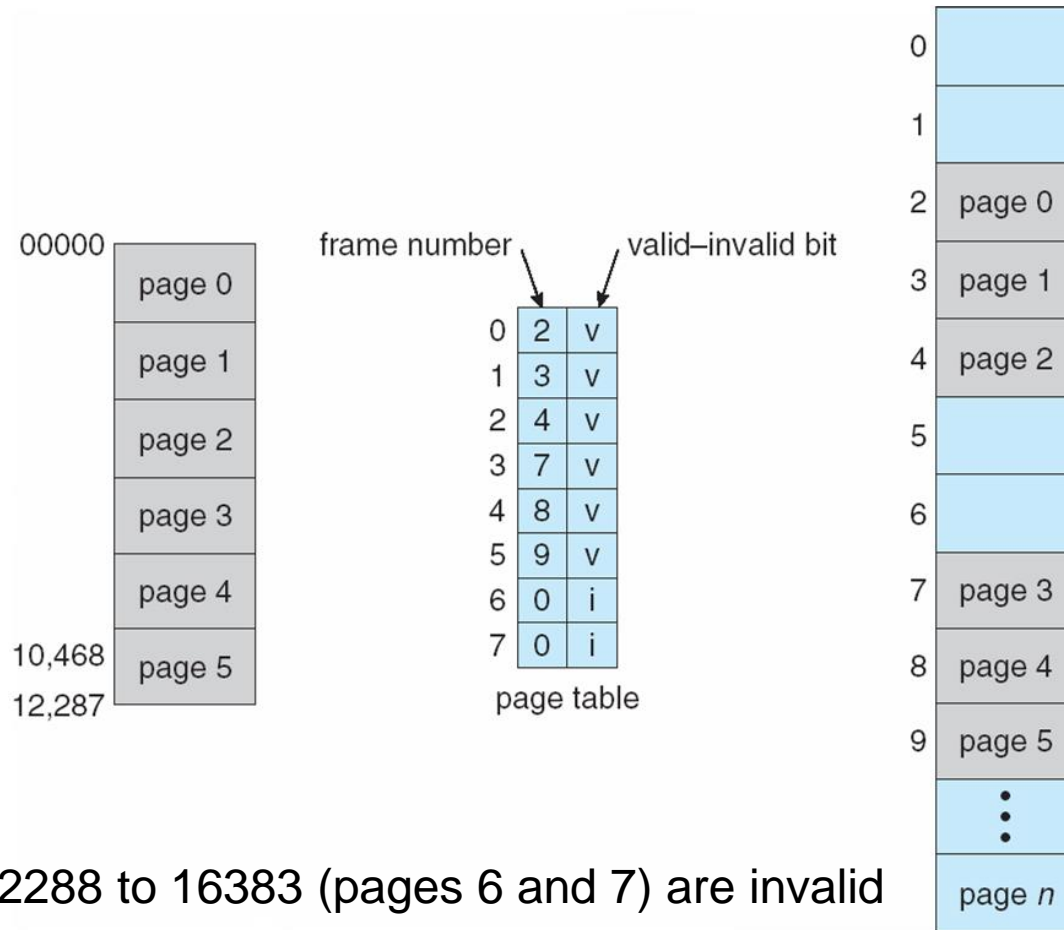
implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,
$$\text{EAT} = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$

implying only 1% slowdown in access time.

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
 - Or use **page-table length register (PTLR)**
 - ▶ Processes use the entire memory very rarely
- Any violations result in a trap to the kernel

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



- Accesses 12288 to 16383 (pages 6 and 7) are invalid

Shared Pages

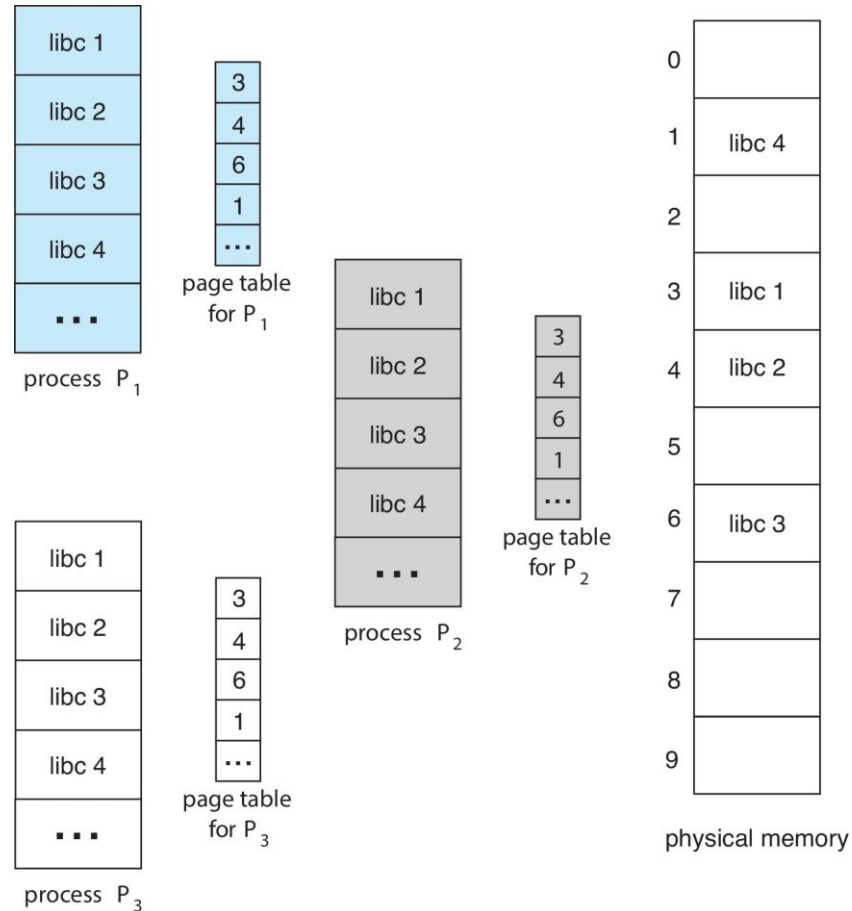
■ Shared code

- 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



Q/A



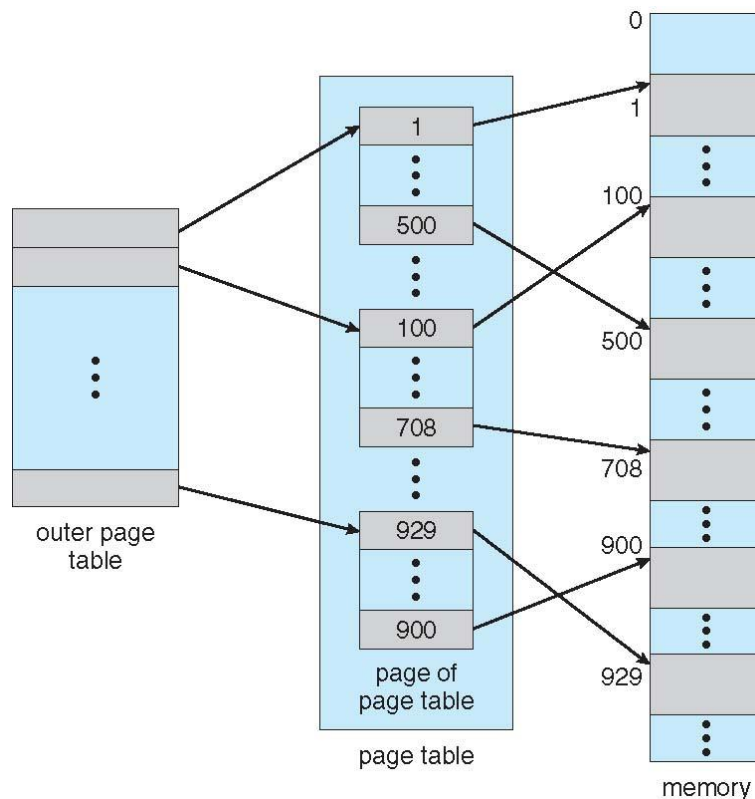
Structure of the Page Table

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 → each process 4 MB of physical address space for the page table alone
 - ▶ Don't want to allocate that contiguously in main memory
 - One simple solution is to divide the page table into smaller units
 - ▶ Hierarchical Paging
 - ▶ Hashed Page Tables
 - ▶ Inverted Page Tables

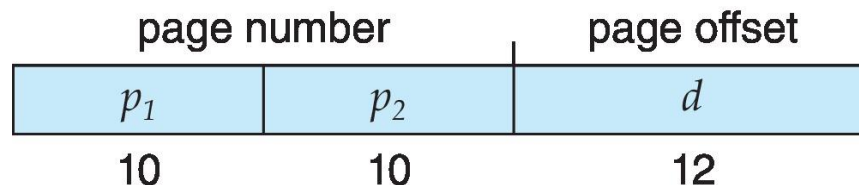
Hierarchical Page Tables

- 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



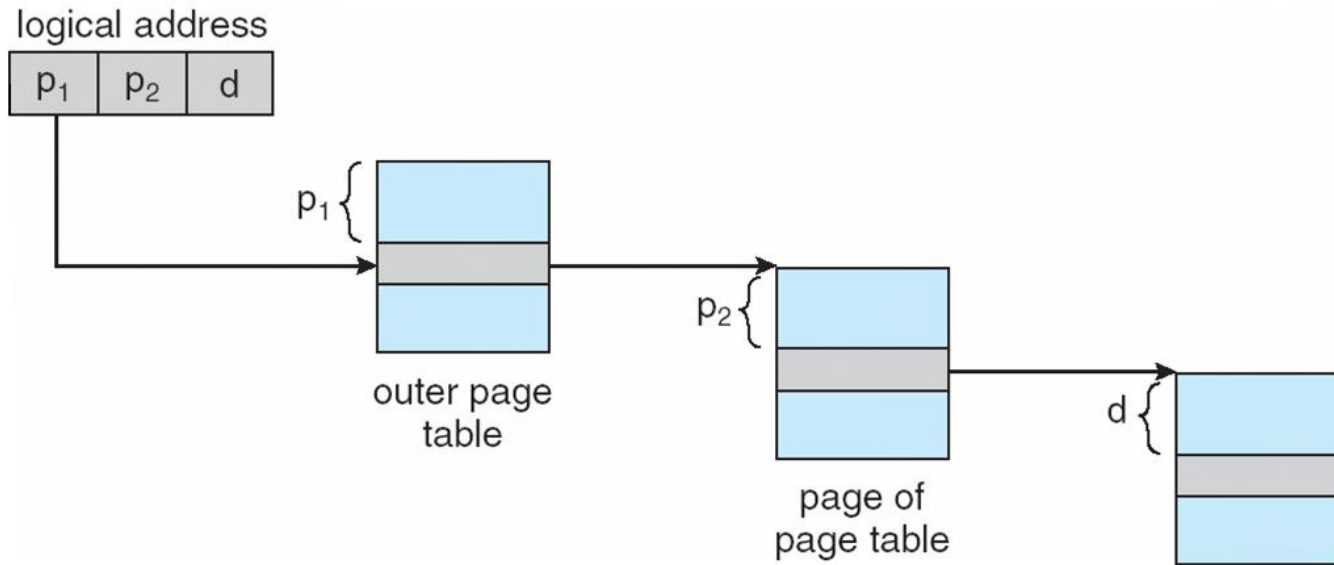
Two-Level Paging Example

- 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 10-bit page number
 - a 12-bit page offset
- Thus, a logical address is as follows:



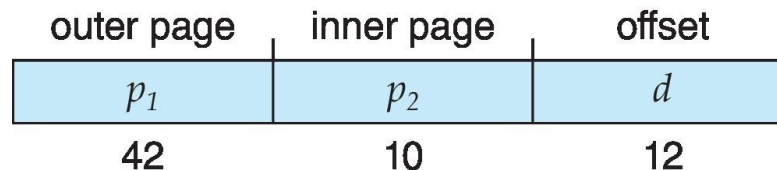
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



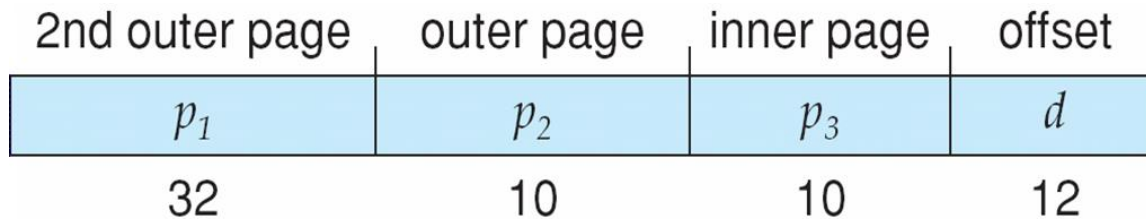
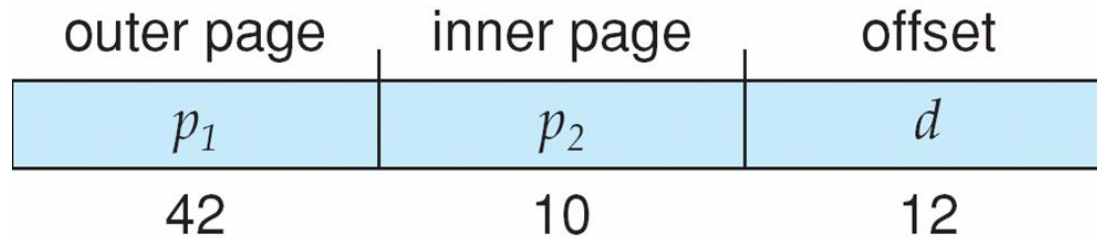
64-bit Logical Address Space

- 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
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location

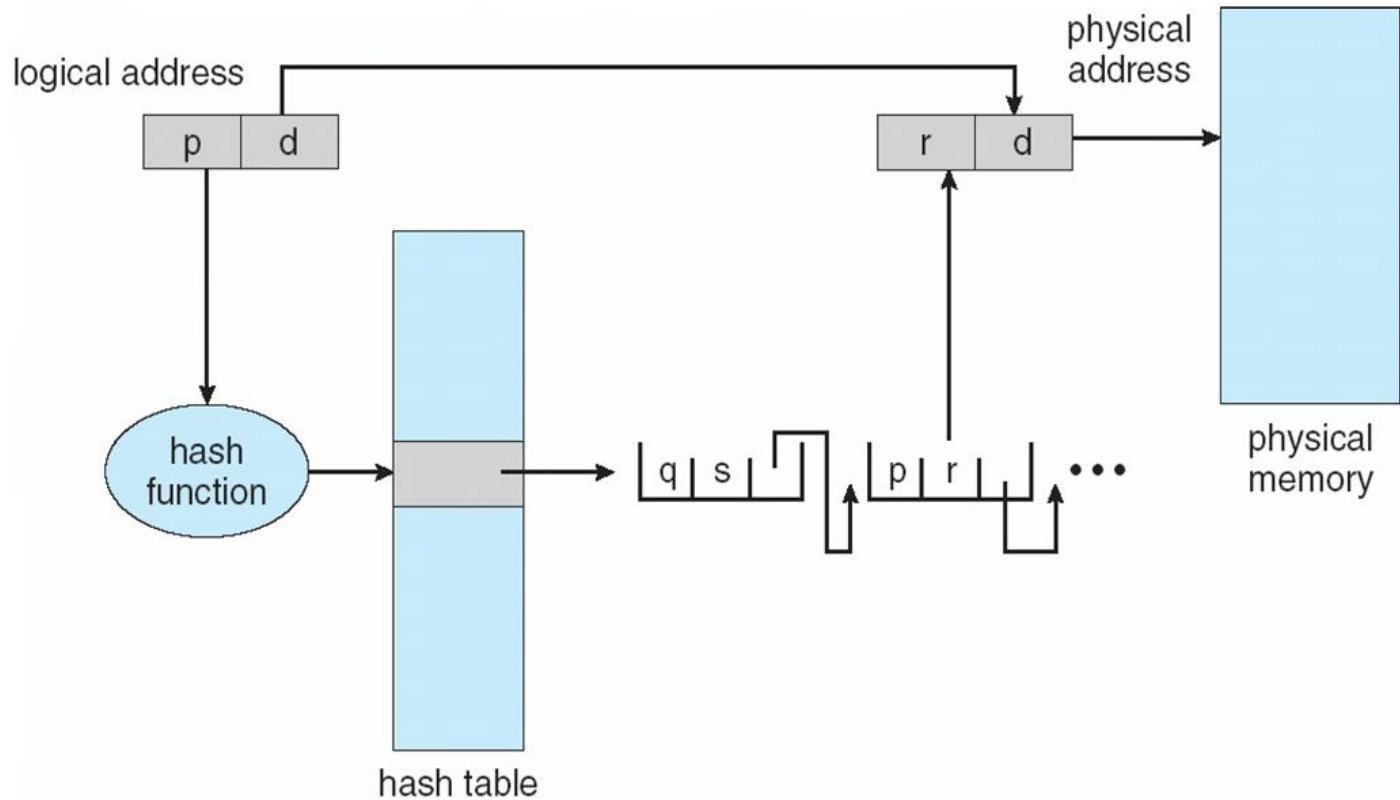
Three-level Paging Scheme



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 (see the figure on the next slide)
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

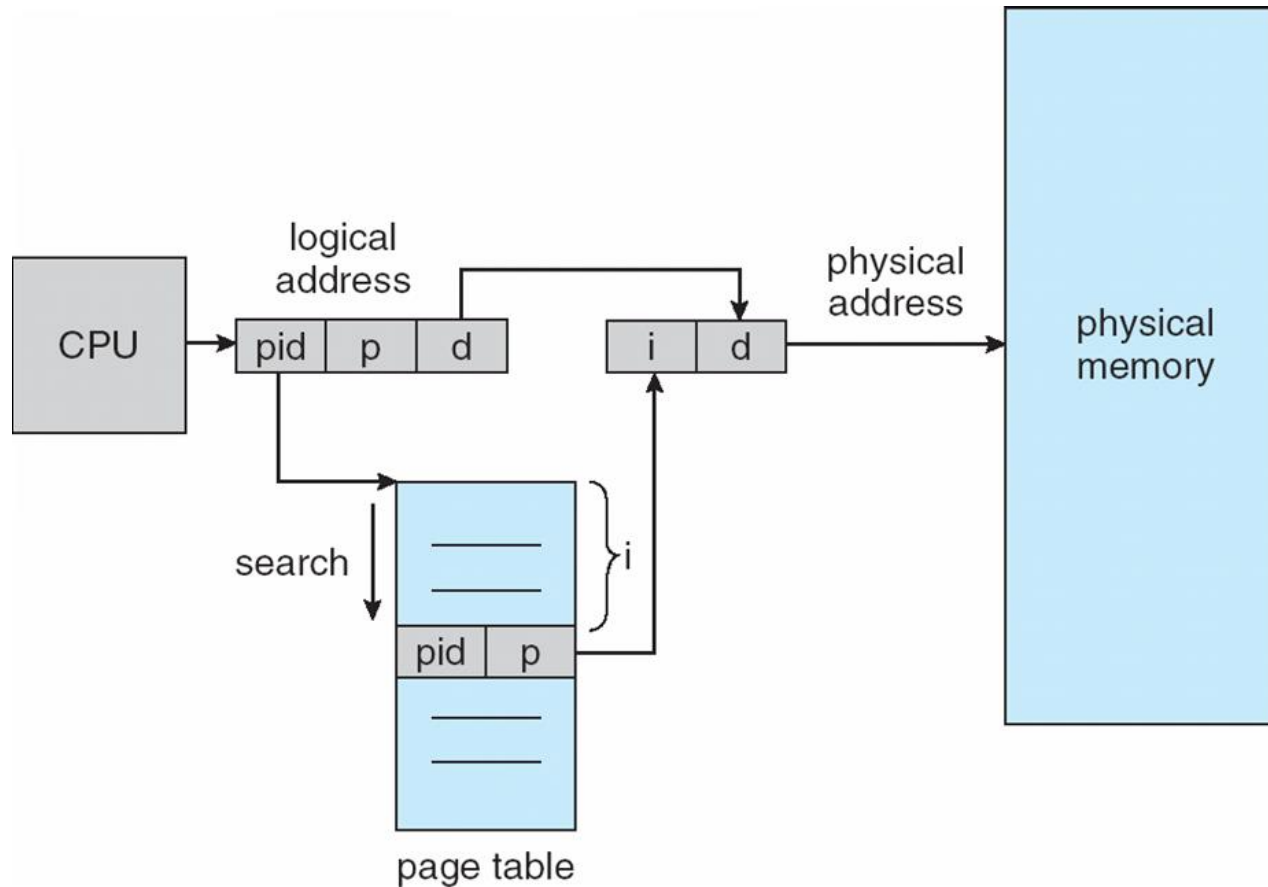
Hashed Page Table



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



Q/A



Swapping

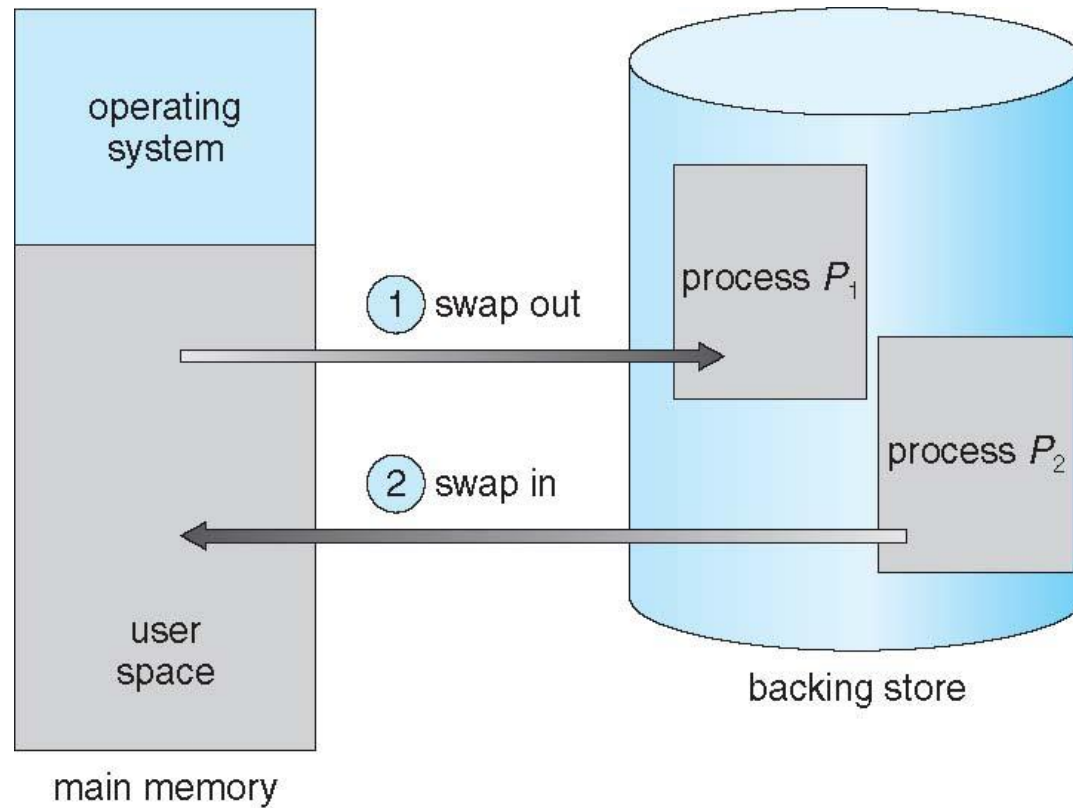
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
 - 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; must provide direct access to these memory images
- **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 (Cont.)

- 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 / from process memory space
- Modified versions of swapping are 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

Schematic View of Swapping



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()`

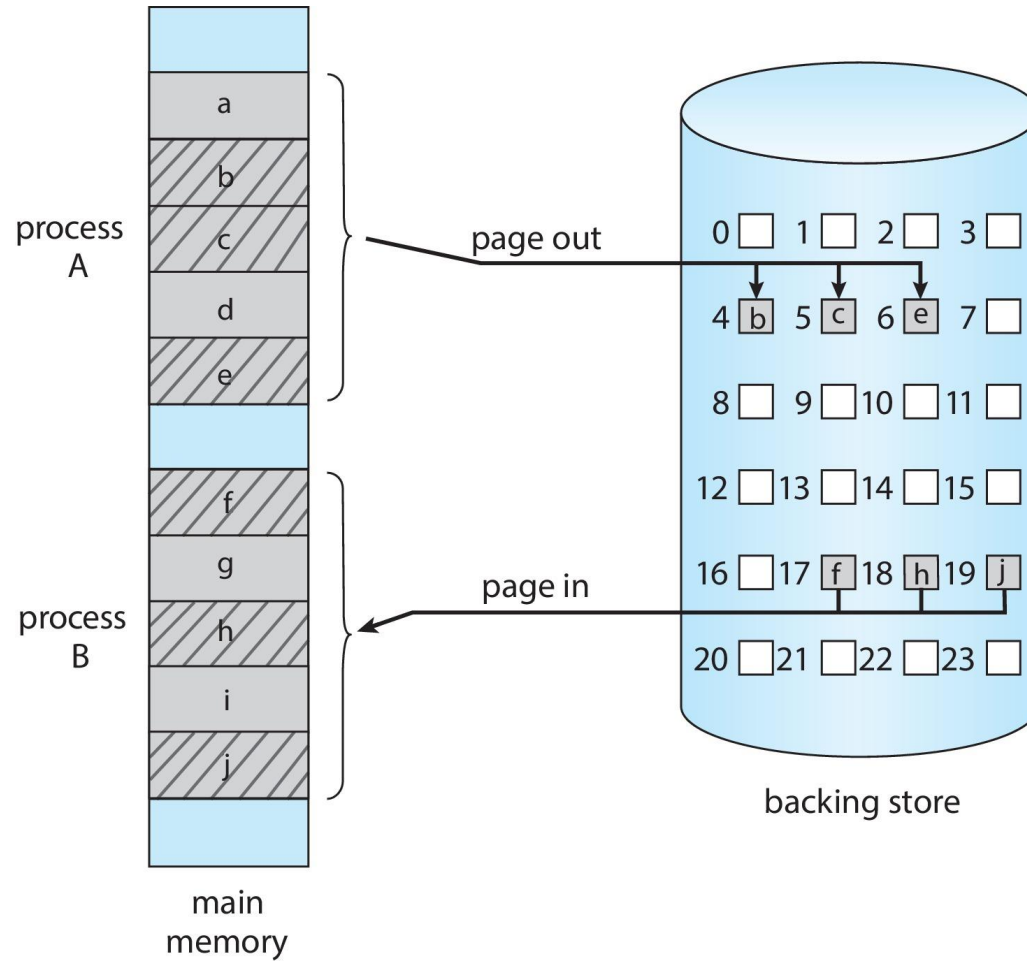
Context Switch Time and Swapping (Cont.)

- 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 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

Swapping with Paging



Example: Intel 32- and 64-bit Architectures

Example: The Intel 32 and 64-bit Architectures

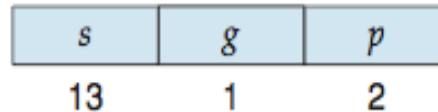
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: 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)**)

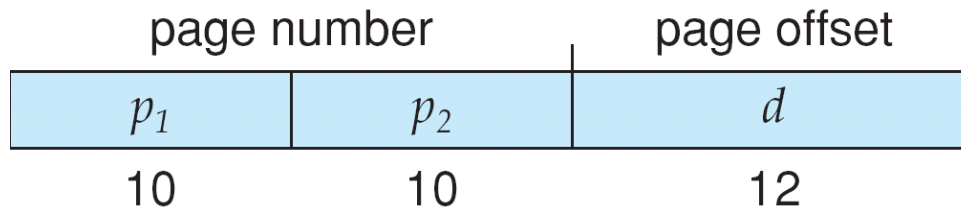
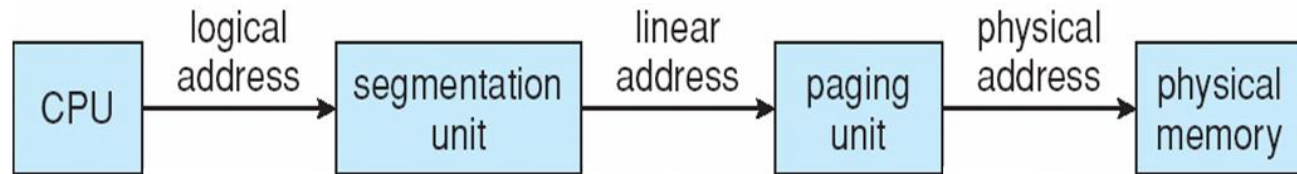
Example: The Intel IA-32 Architecture (Cont.)

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

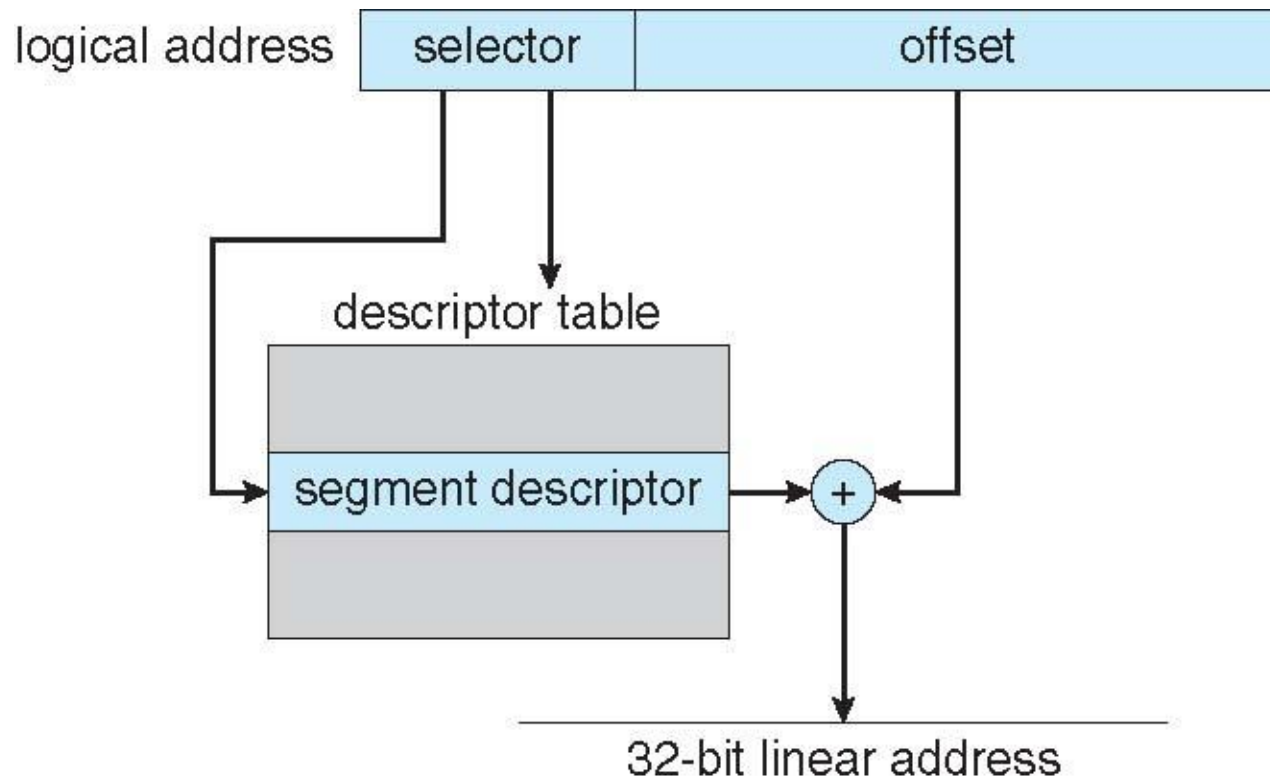


- Linear address given to paging unit
 - ▶ Which generates physical address in main memory
 - ▶ Paging units form equivalent of MMU
 - ▶ Pages sizes can be 4 KB or 4 MB

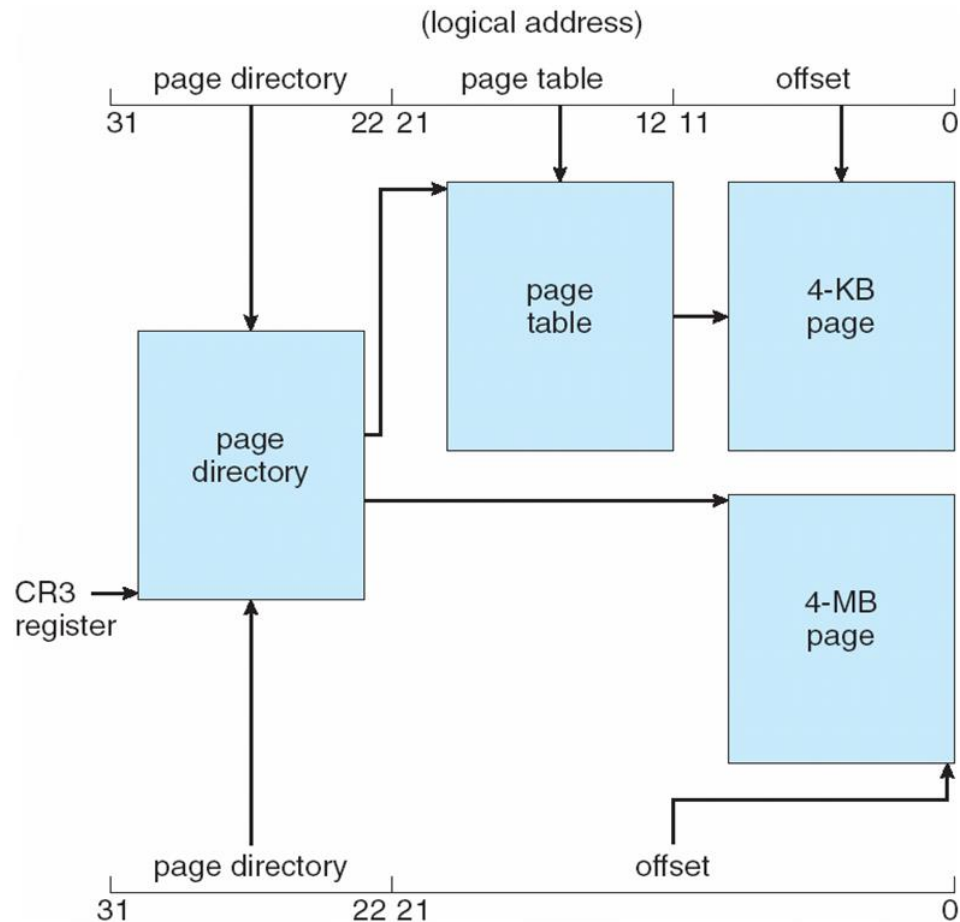
Logical to Physical Address Translation in IA-32



Intel IA-32 Segmentation

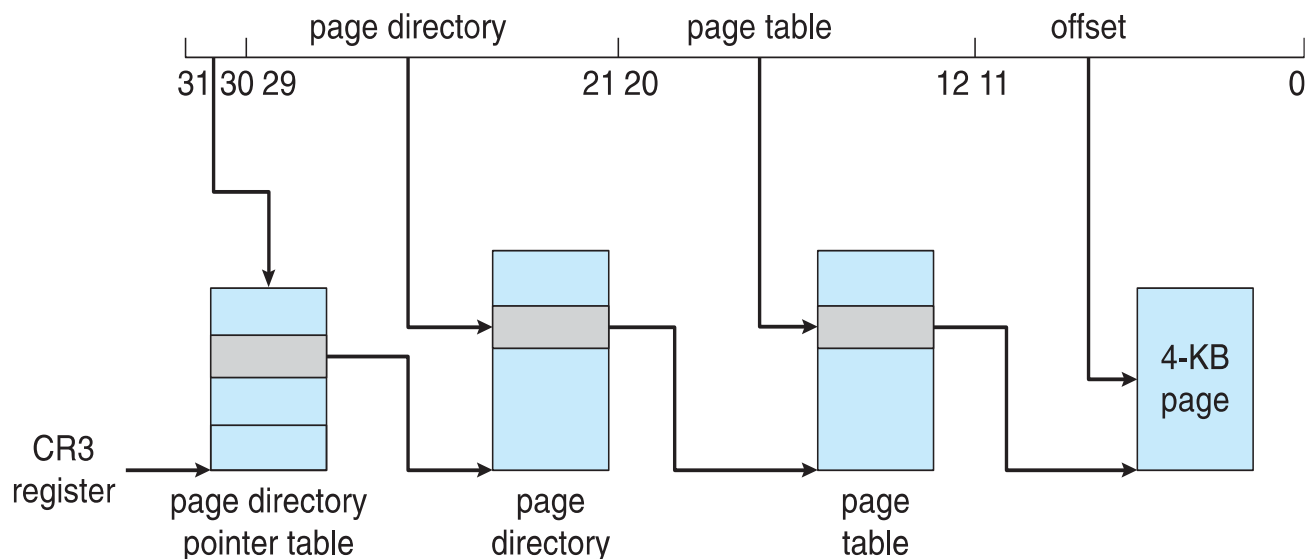


Intel IA-32 Paging Architecture



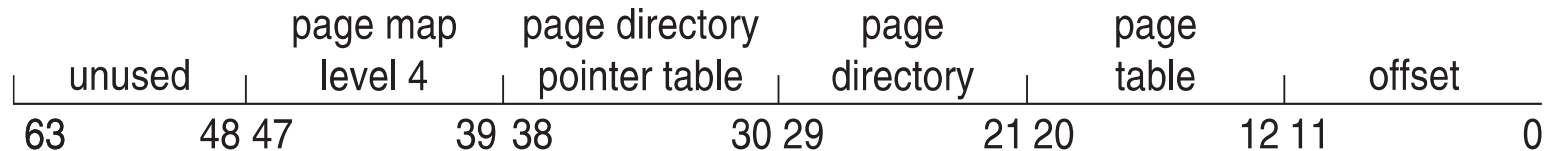
Intel IA-32 Page Address Extensions

- 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



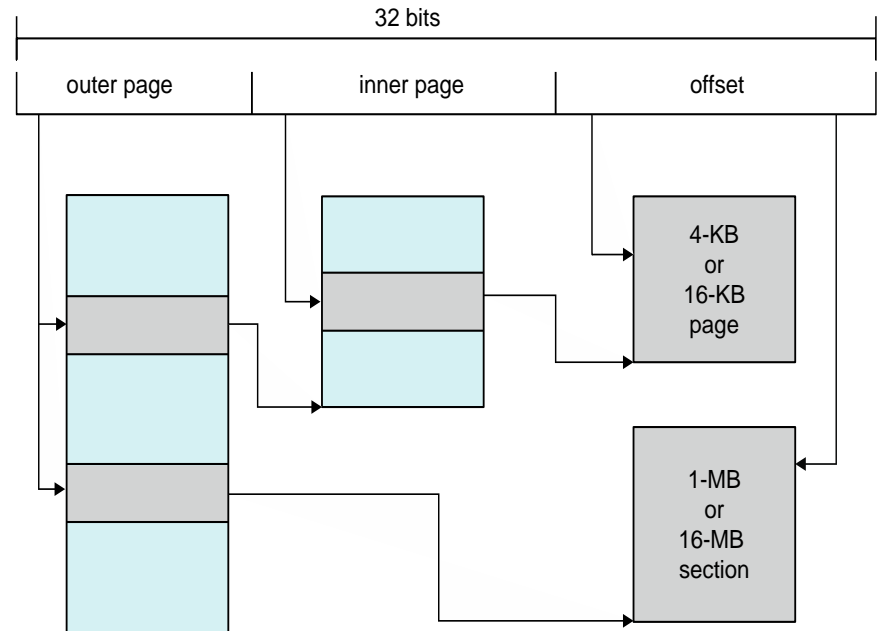
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
 - Page sizes of 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)
- 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



Q/A

