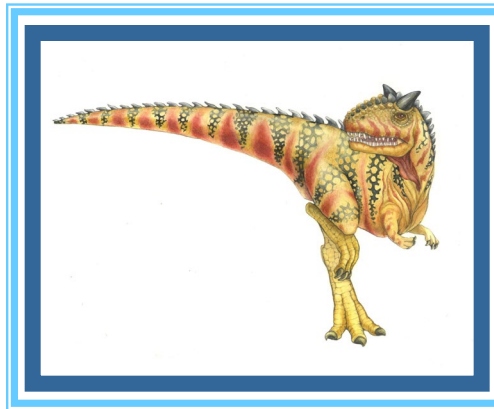


Chapter 9

Main Memory (2)





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)**
- Any violations result in a trap to the kernel





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

00000

page 0
page 1
page 2
page 3
page 4
page 5

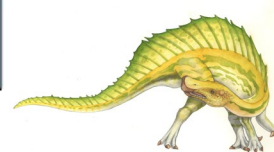
10,468
12,287

frame number valid-invalid bit

0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page n





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

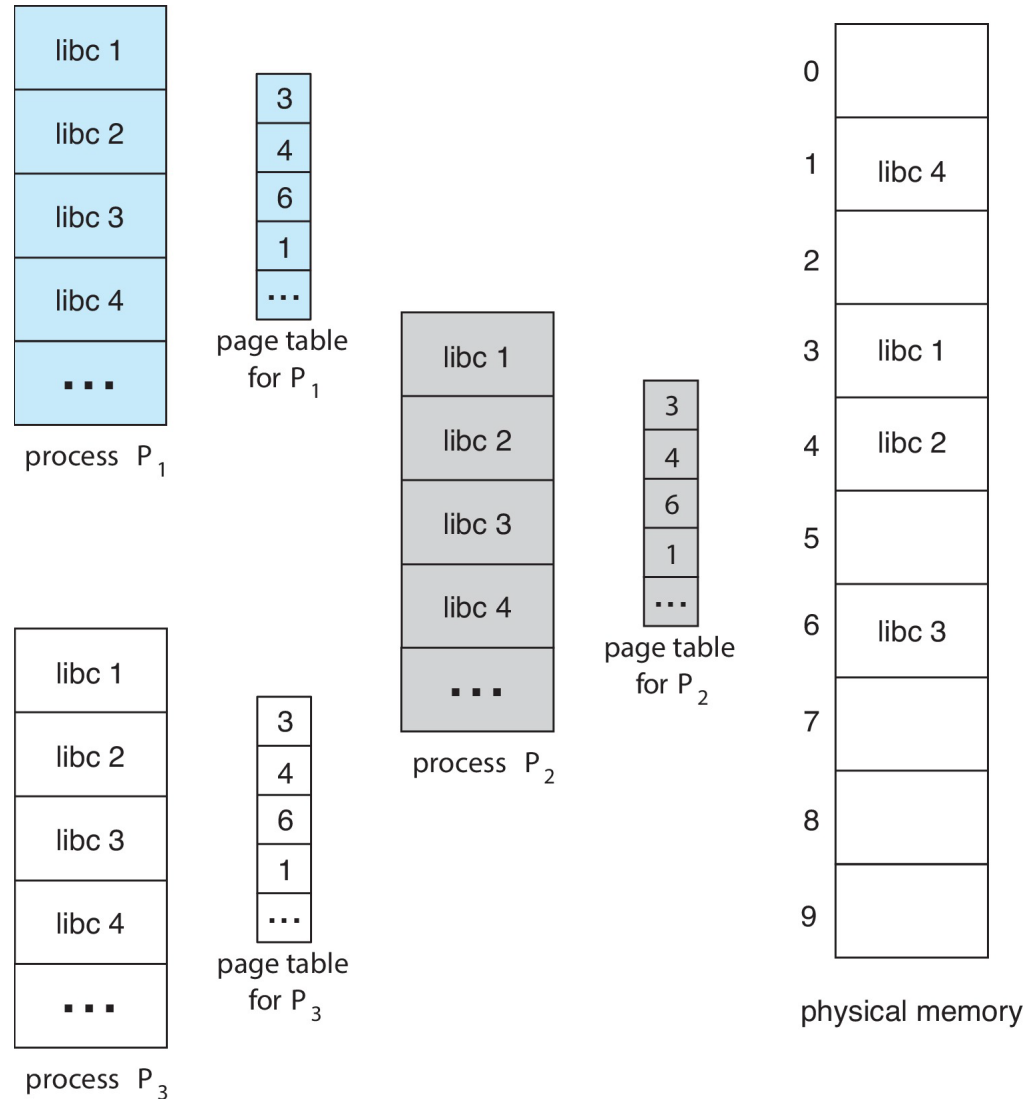
■ Private code and data

- 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

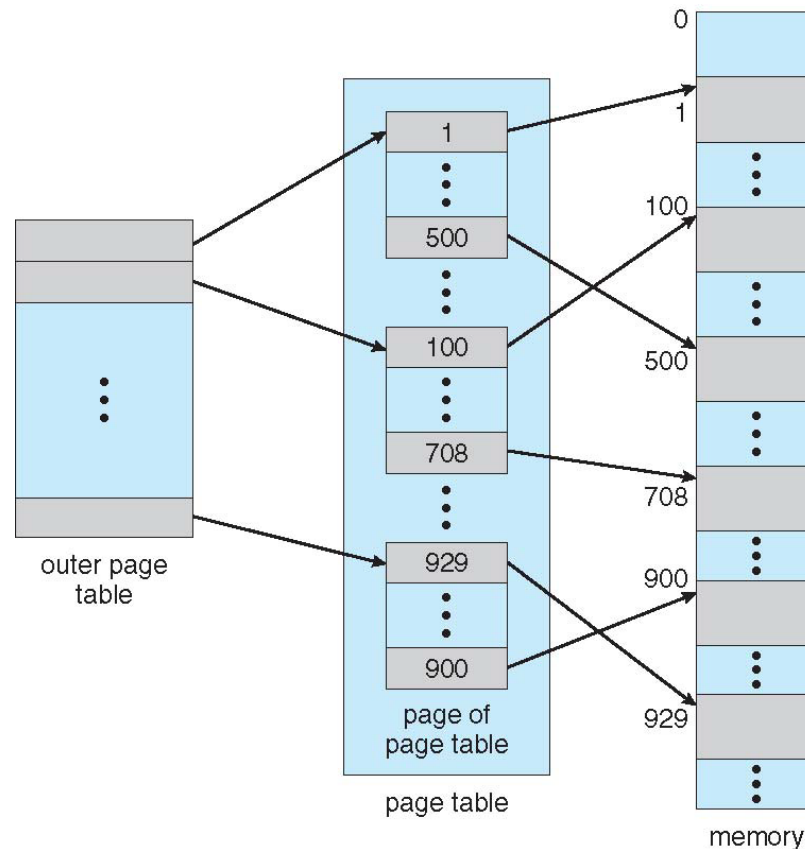
- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes → each process 4 MB of physical address 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

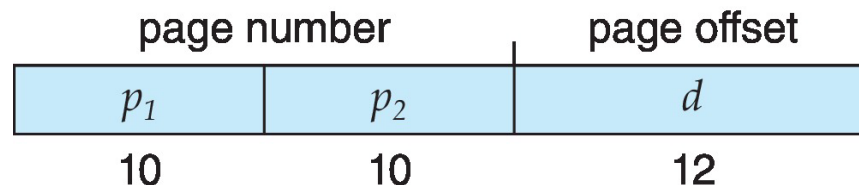
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



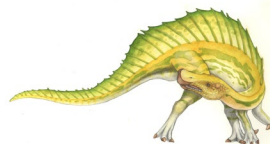


Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - a 10-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:

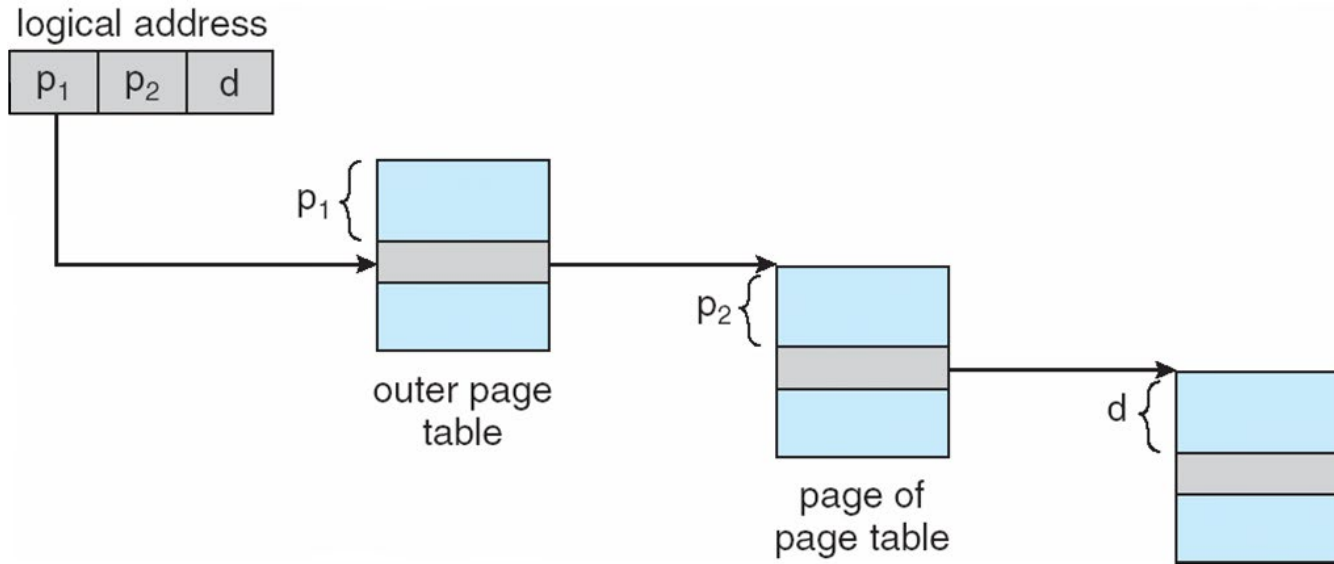


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





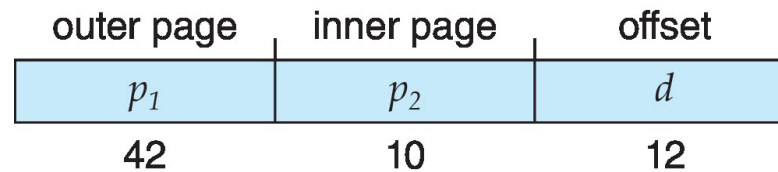
Address-Translation Scheme



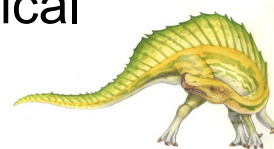


64-bit Logical Address Space

- 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

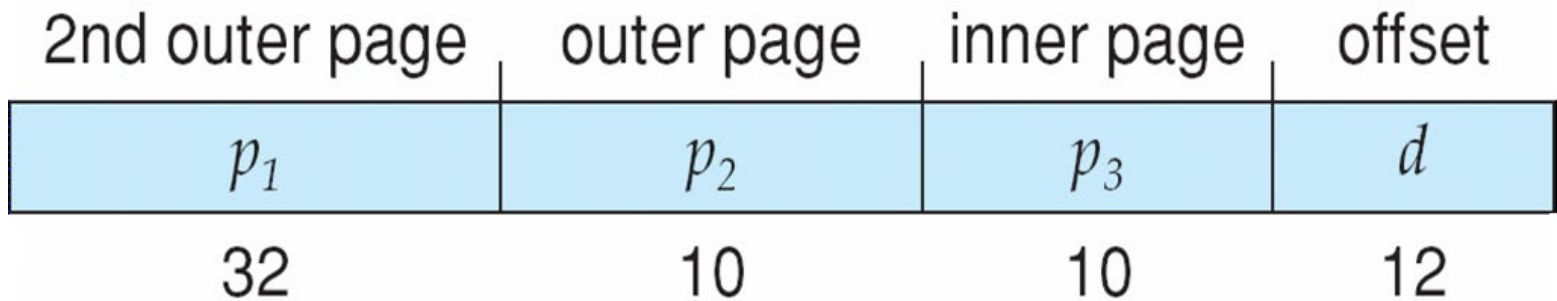
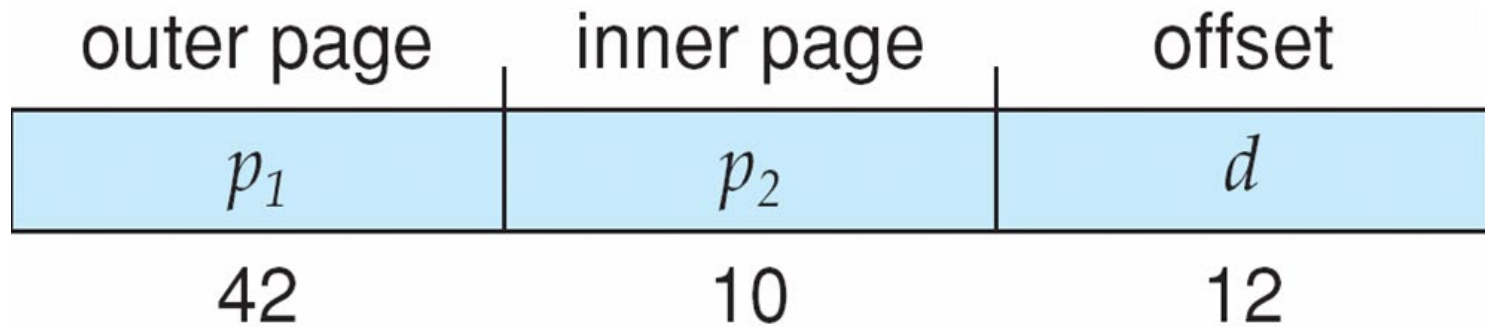


- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2^{nd} outer page table
- But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location





Three-level Paging Scheme





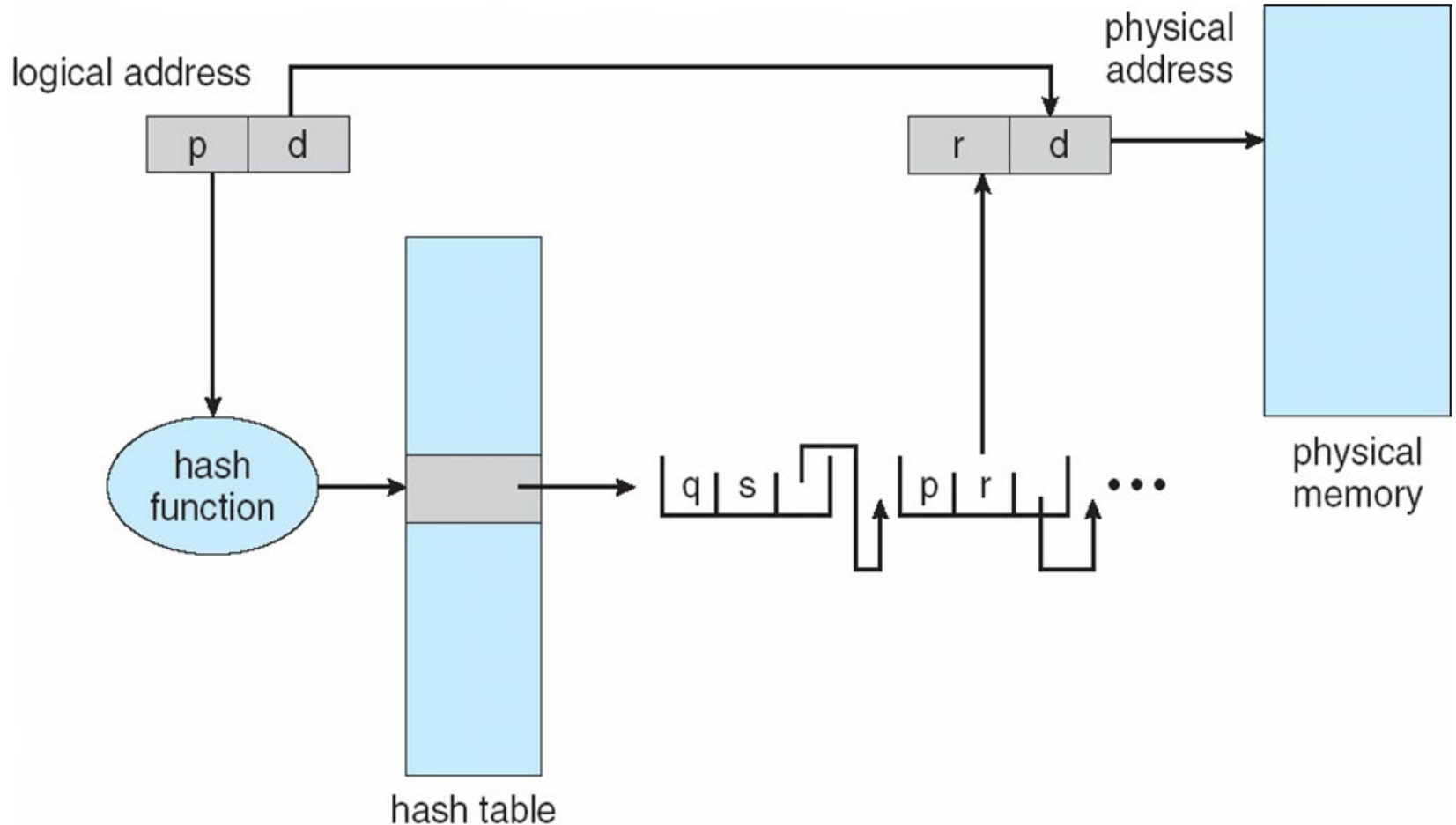
Hashed Page Tables (1)

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)





Hashed Page Table (2)





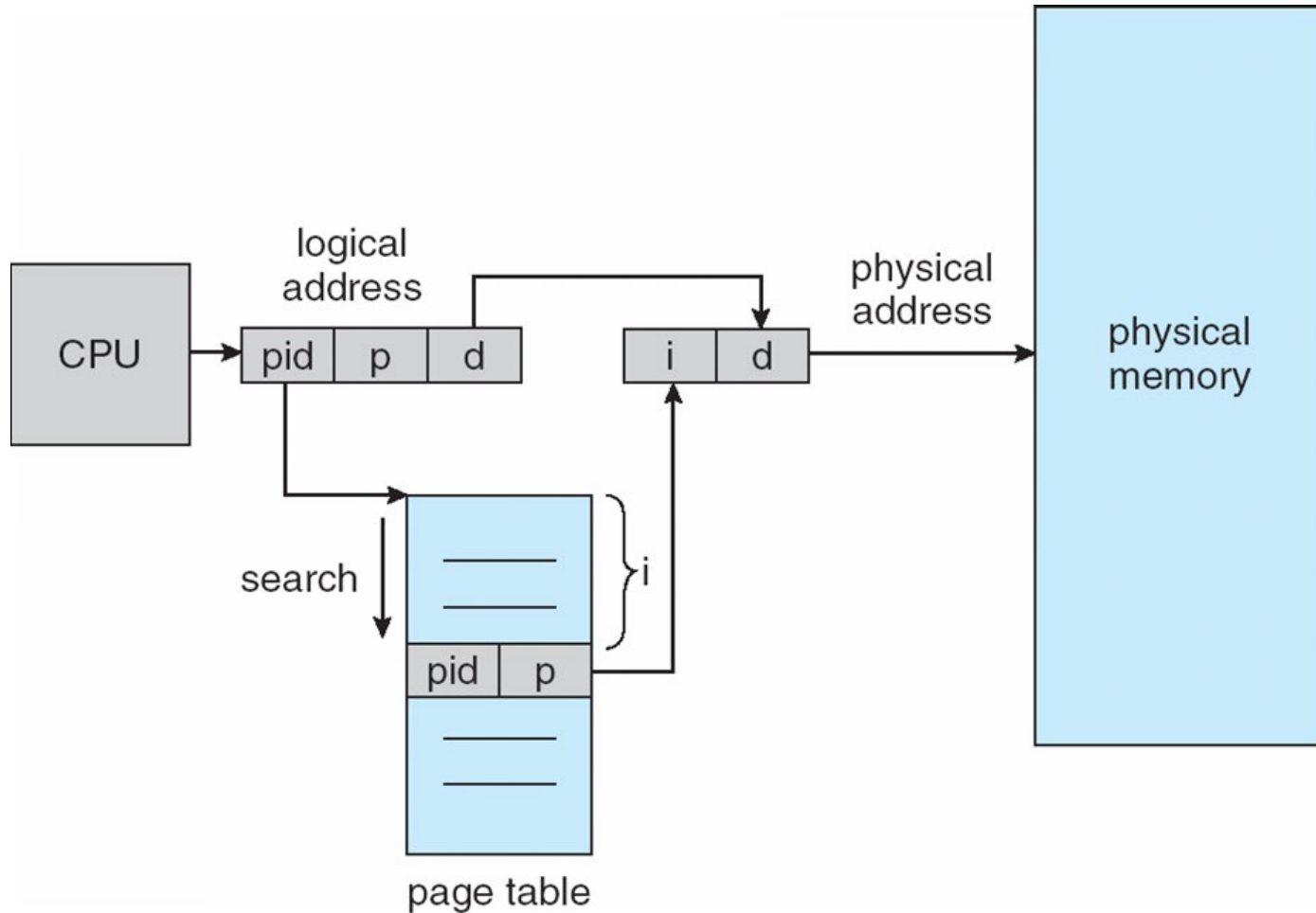
Inverted Page Table

- 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





Oracle SPARC Solaris (1)

- Consider modern, 64-bit operating system 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 (2)

- 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.





Swapping (1)

- 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





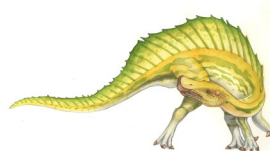
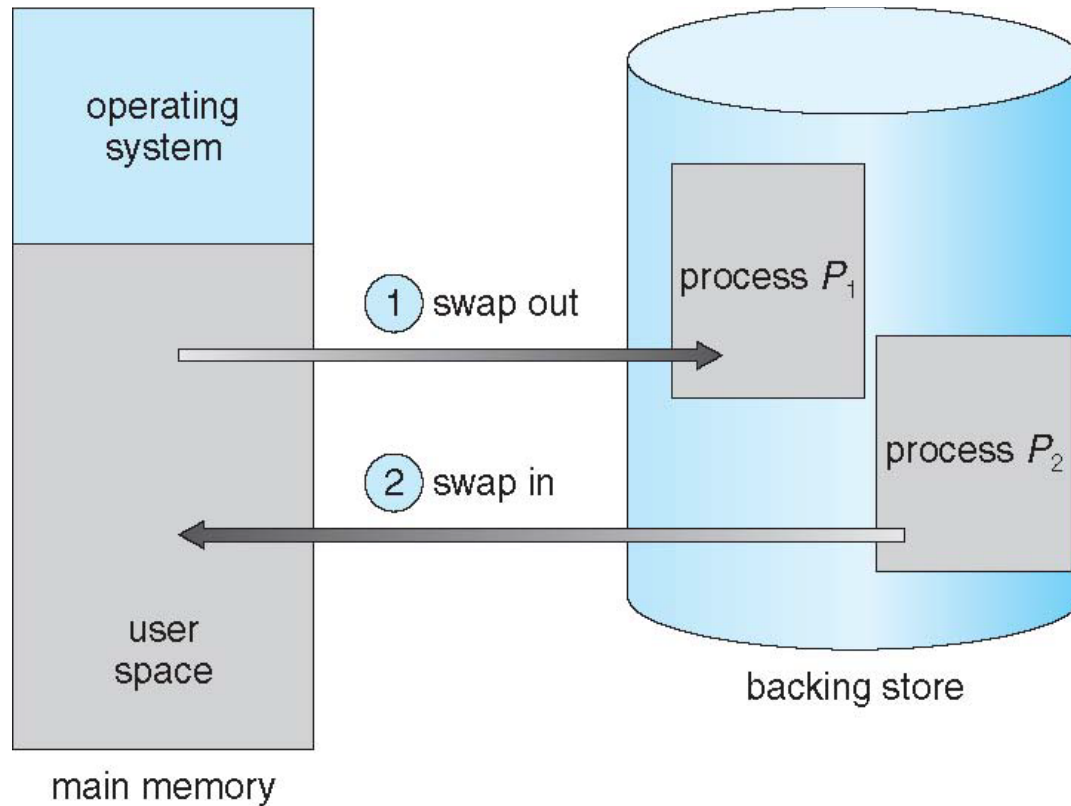
Swapping (2)

- 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 and Swapping (1)

- 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 (2)

- 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





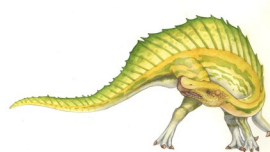
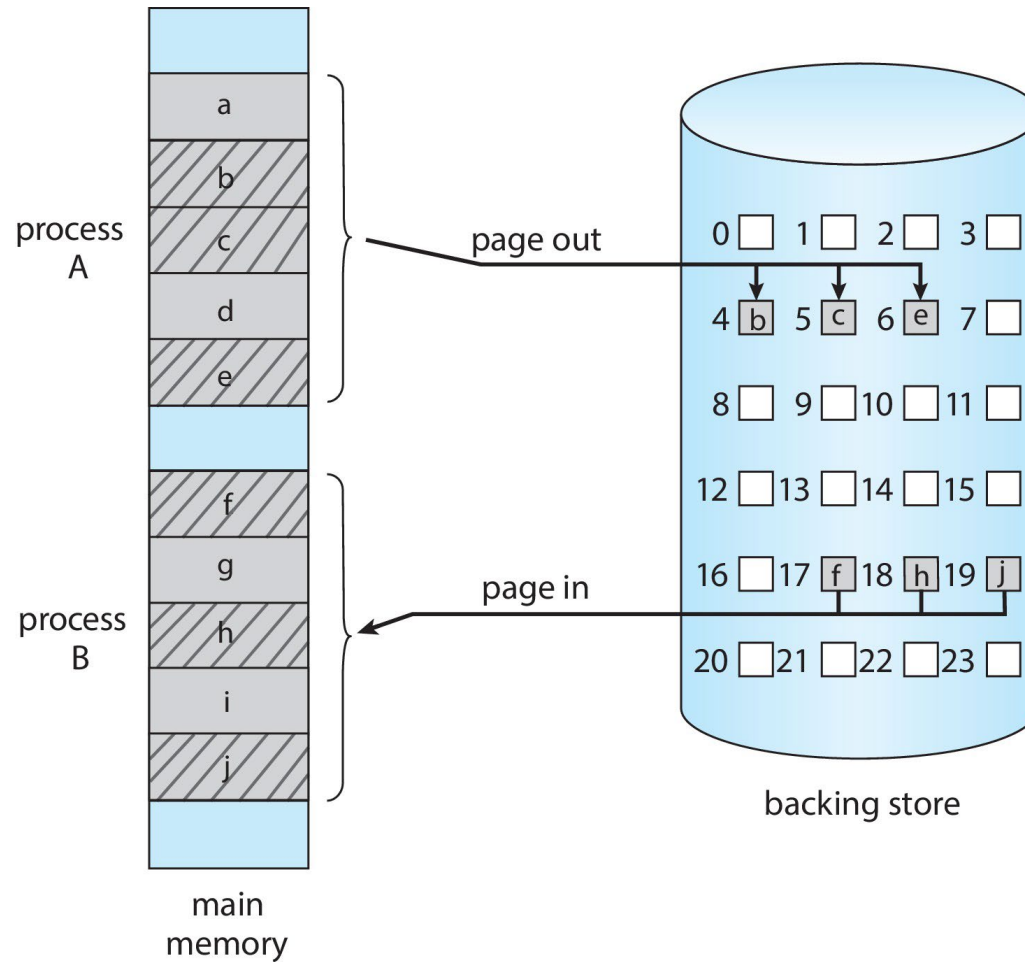
Swapping on Mobile Systems

- 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





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





The Intel IA-32 Architecture (1)

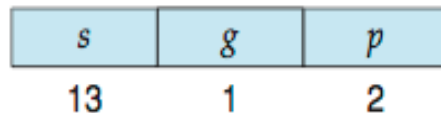
- Supports both segmentation and segmentation with paging
 - Each segment can be 4 GB
 - Up to 16 K segments per process
 - Divided into two partitions
 - ▶ First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
 - ▶ Second partition of up to 8K segments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





The Intel IA-32 Architecture (2)

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

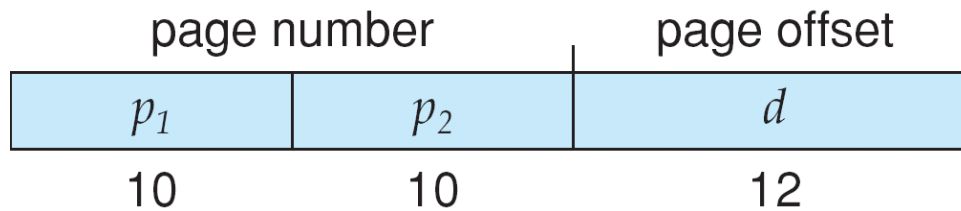
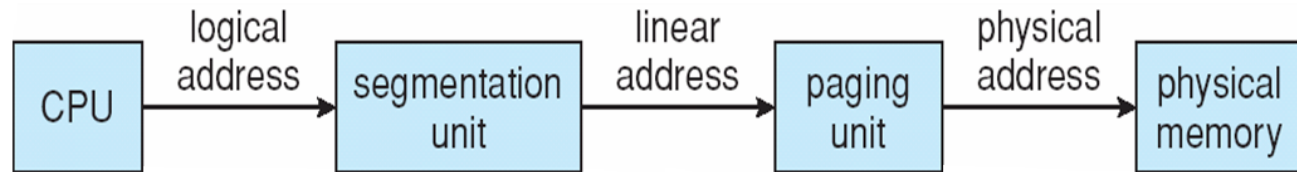


- Linear address given to paging unit
 - ▶ Which generates physical address in main memory
 - ▶ Paging units form equivalent of MMU
 - ▶ Pages sizes can be 4 KB or 4 MB



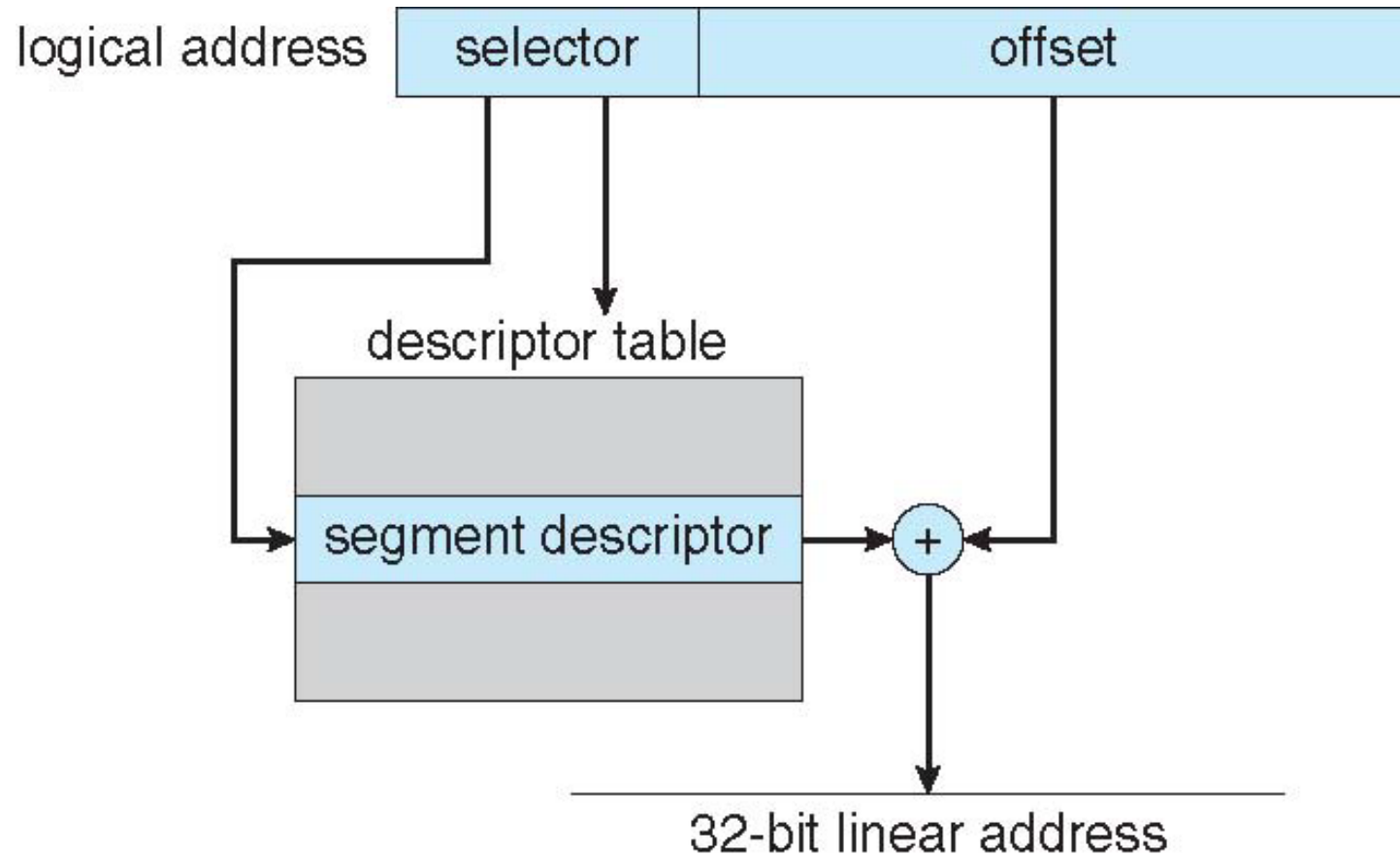


Logical to Physical Address Translation in IA-32



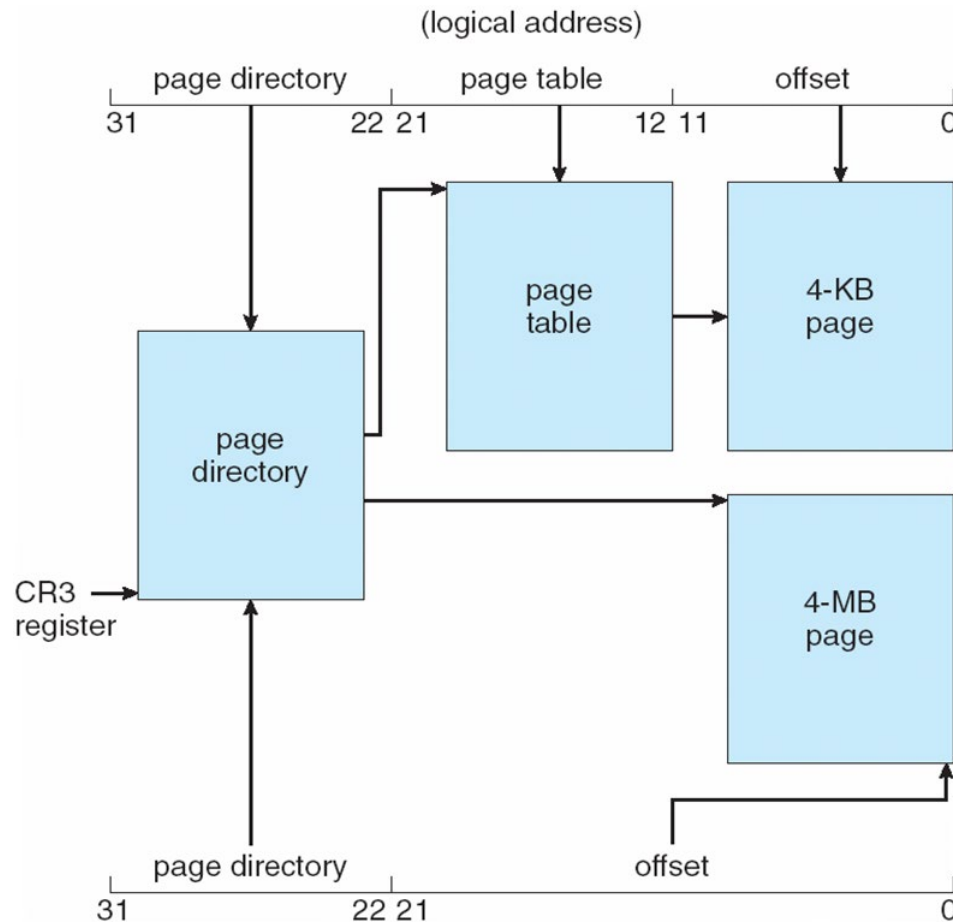


Intel IA-32 Segmentation





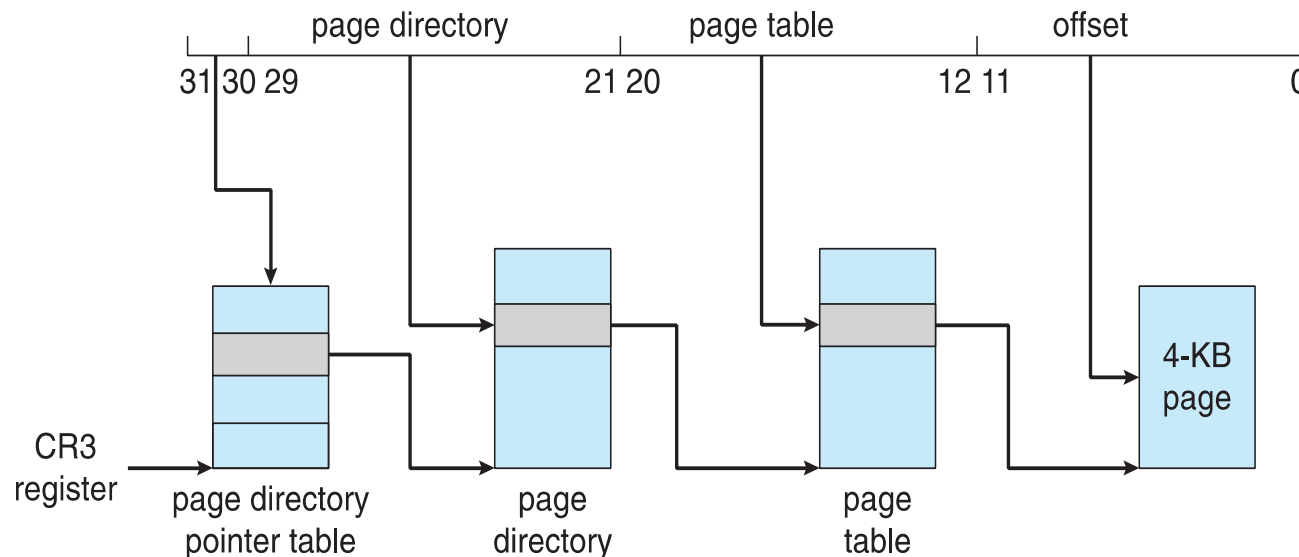
Intel IA-32 Paging Architecture





Intel IA-32 Page Address Extensions

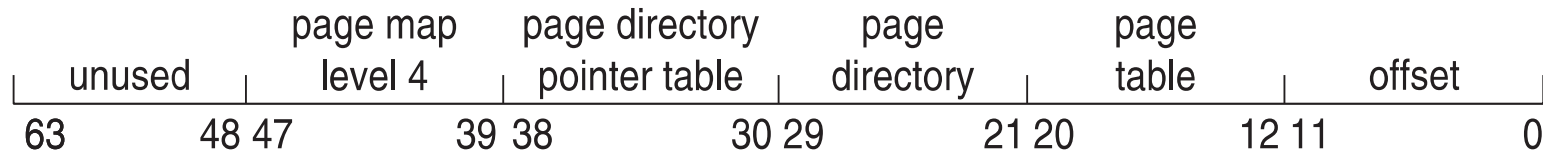
- 32-bit address limits led Intel to create **page address extension (PAE)**
 - allowing 32-bit apps access to more than 4GB of memory space
 - Paging went to a 3-level scheme
 - Top two bits refer to a **page directory pointer table**
 - Page-directory and page-table entries moved to 64-bits in size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





Intel x86-64

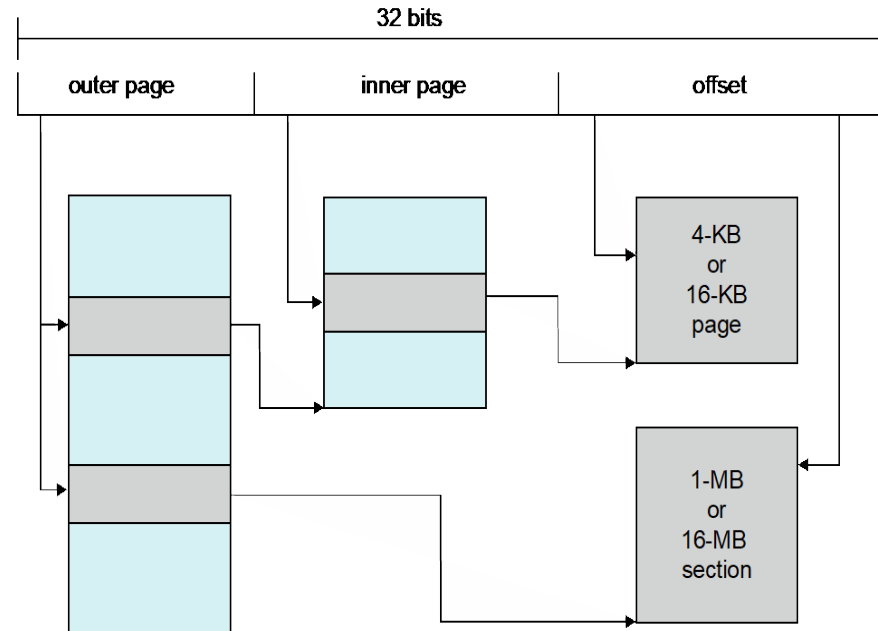
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
 - Page sizes of 4 KB, 2 MB, 1 GB
 - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits





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



End of Chapter 9

