

# Chapter 9: Main Memory

---





# Chapter 9: Memory Management

---

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





# Objectives

---

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging





# Background

---

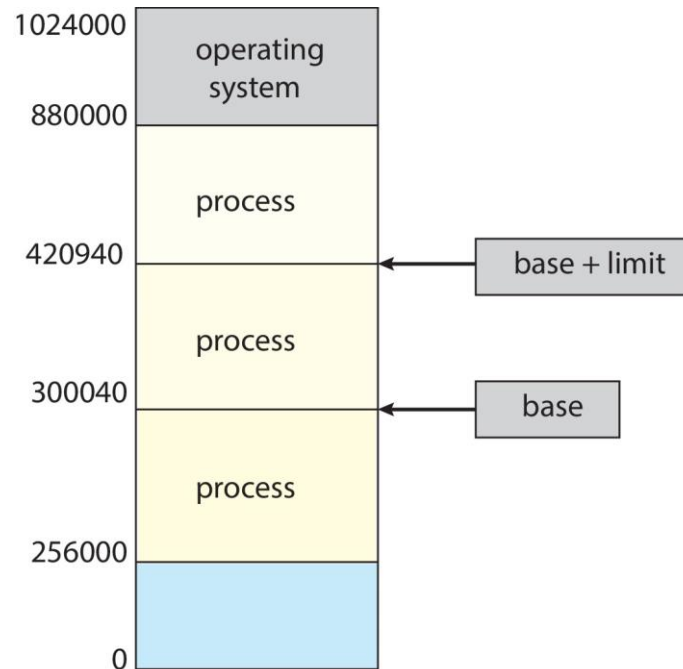
- Program must be brought (from disk) into memory and placed within the context of a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access 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

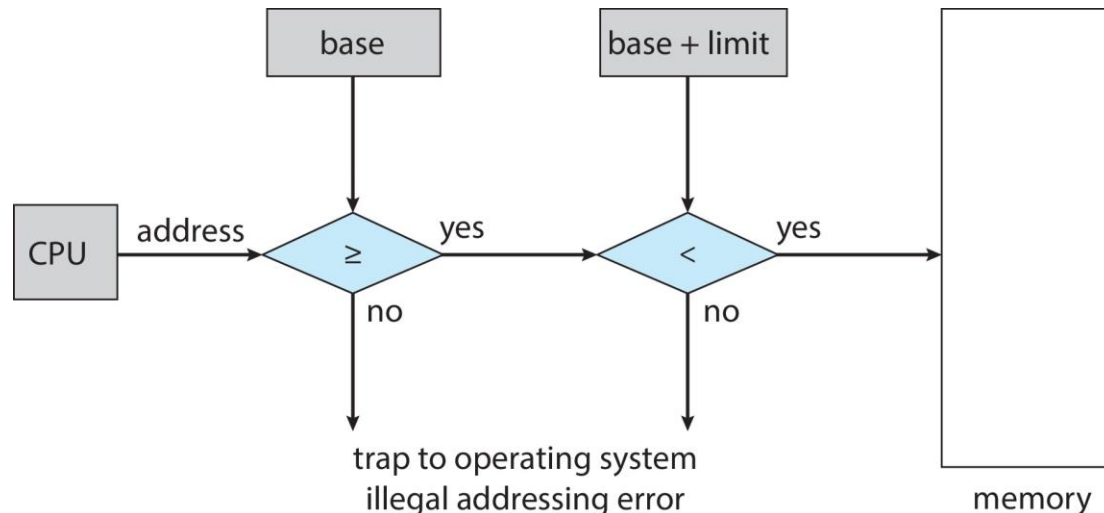
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process





# 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 load the base and limit registers are privileged





# Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Most systems allow a user process to reside in any part of the physical memory instead of 0000
- Addresses are represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - ▶ i.e., “14 bytes from beginning of this module”
  - Linker or loader will bind relocatable addresses to absolute addresses
    - ▶ i.e., 74014
  - Each binding maps one address space to another





# Binding of Instructions and Data to Memory

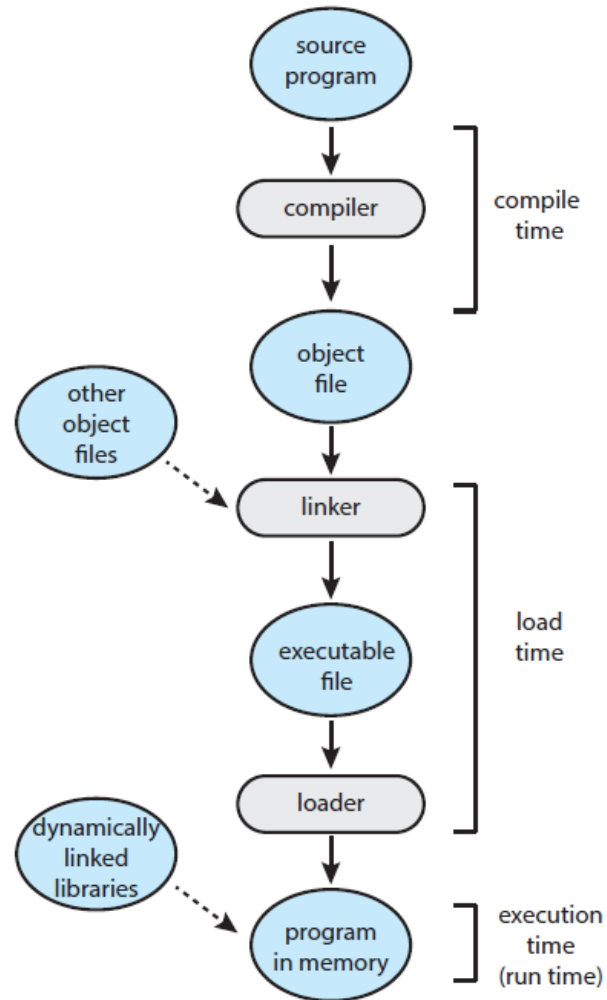
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location is known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time, and resolution happens at load 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)

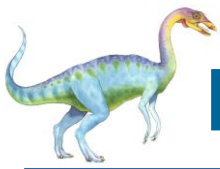






# Multistep Processing of a User Program





# Logical vs. Physical Address Space

---

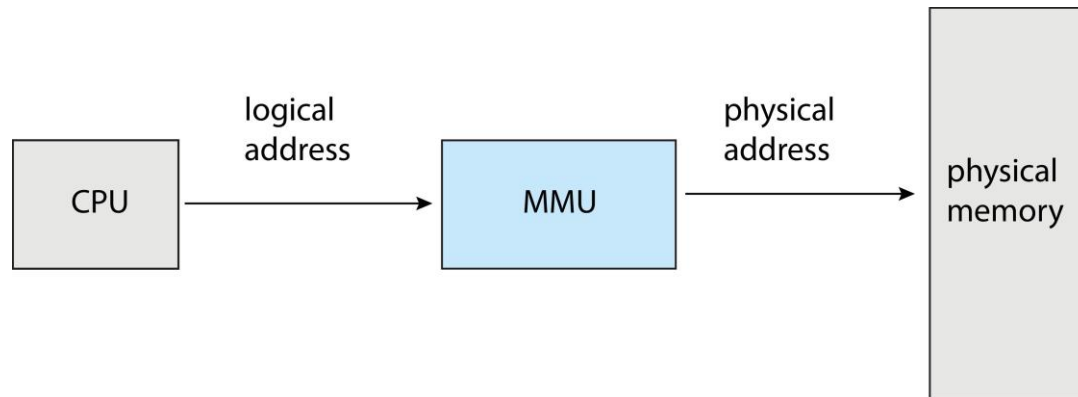
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **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



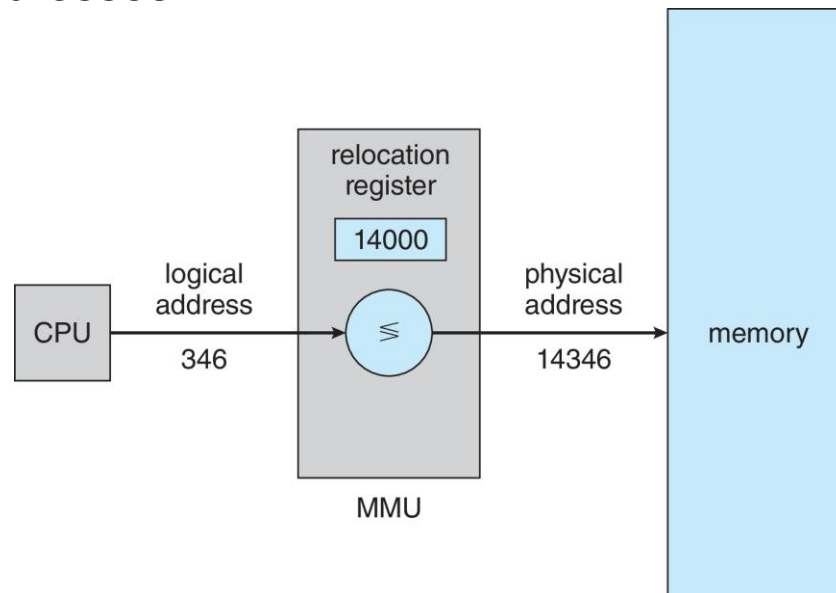
- Many methods possible, covered in the rest of this chapter





# 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
- The user program deals with *logical* addresses; it never sees the *real* physical addresses





# Dynamic Loading

---

- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading





# Dynamic Linking

- **Static linking** – system libraries and program code combined by 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
- 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 libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed





# 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
    - ▶ Many OS (including Linux and Windows) place the OS code in high memory instead
  - Each process contained in single contiguous section of memory





# Contiguous Allocation (Cont.)

---

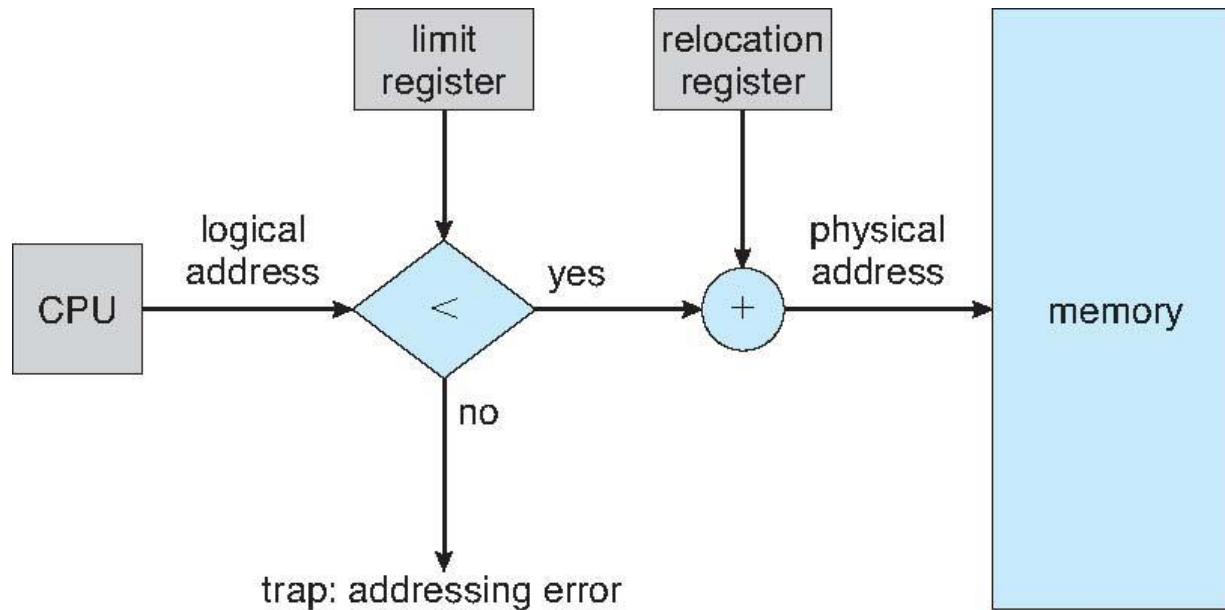
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - The relocation (base) and limit registers are per-process and loaded during context switch
  - MMU maps logical address *dynamically*
  - This scheme allows the OS' size to change dynamically, i.e., certain module (like a device driver) can be loaded into memory only when it is needed and removed when it is no longer needed.







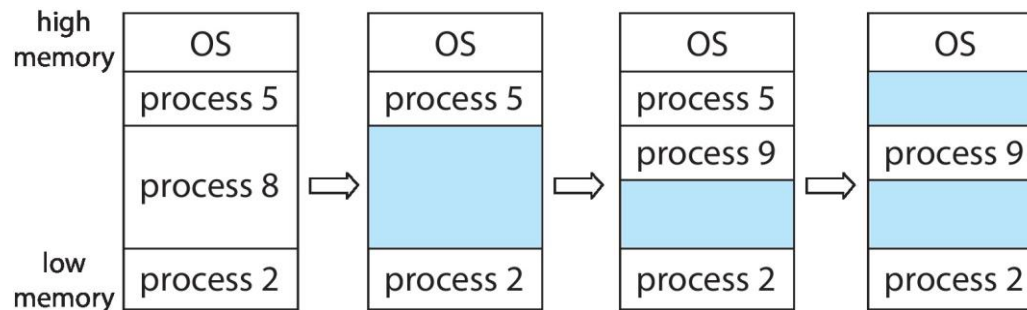
# Hardware Support for Relocation and Limit Registers





# 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

Simulations show that 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
- Statistical analysis of first fit reveals that given  $N$  blocks allocated, another  $0.5 N$  blocks will be lost to fragmentation
  - $1/3$  may be unusable -> known as the **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
  - Then relocation requires only moving the program and data, before changing the base register to reflect the new base address
- This scheme, however, is expensive
- Can we have noncontiguous physical address space?





# Paging

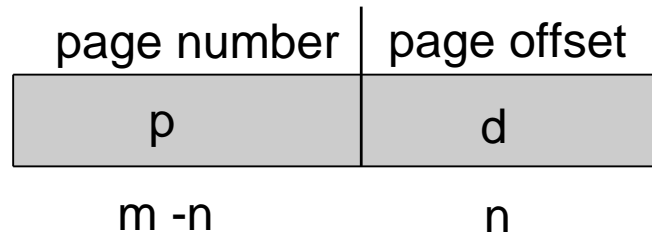
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes ( $2^9$ ) and 16 Mbytes ( $2^{24}$ )
- 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

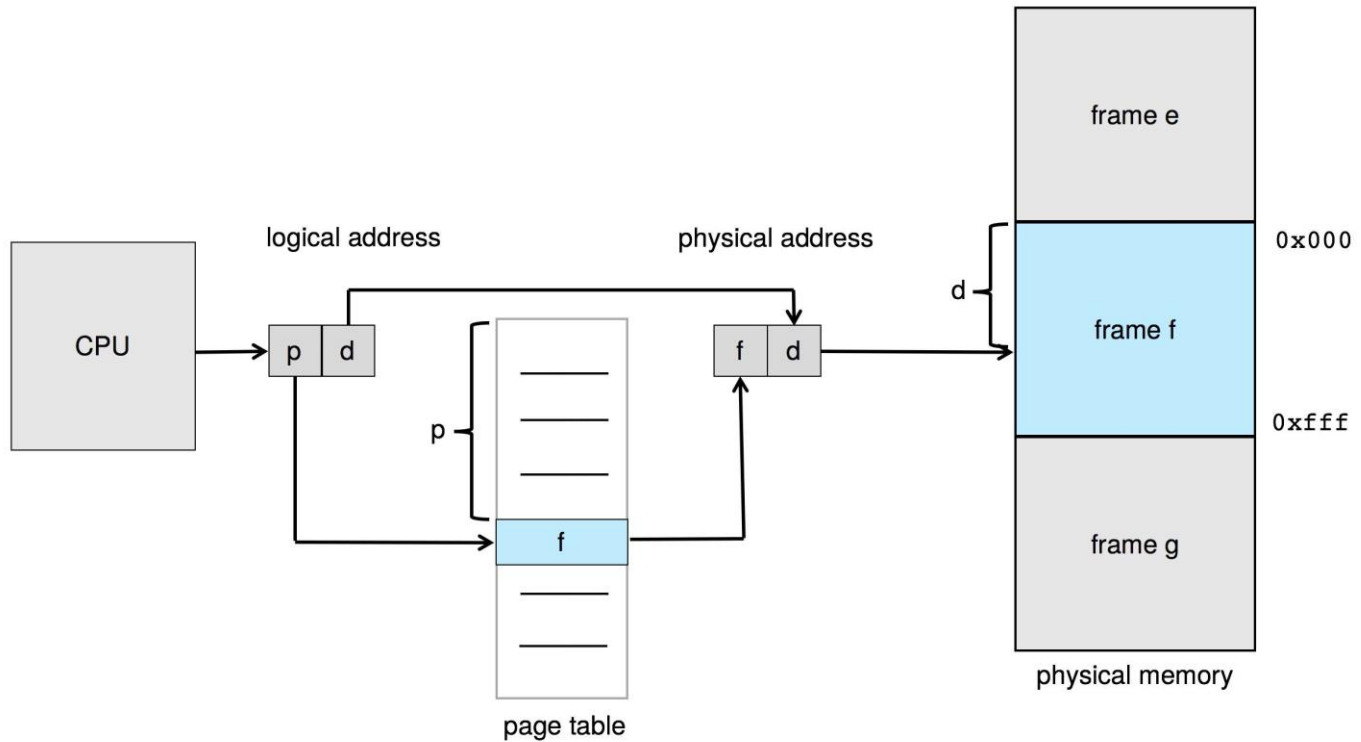


- For given logical address space  $2^m$  and page size  $2^n$





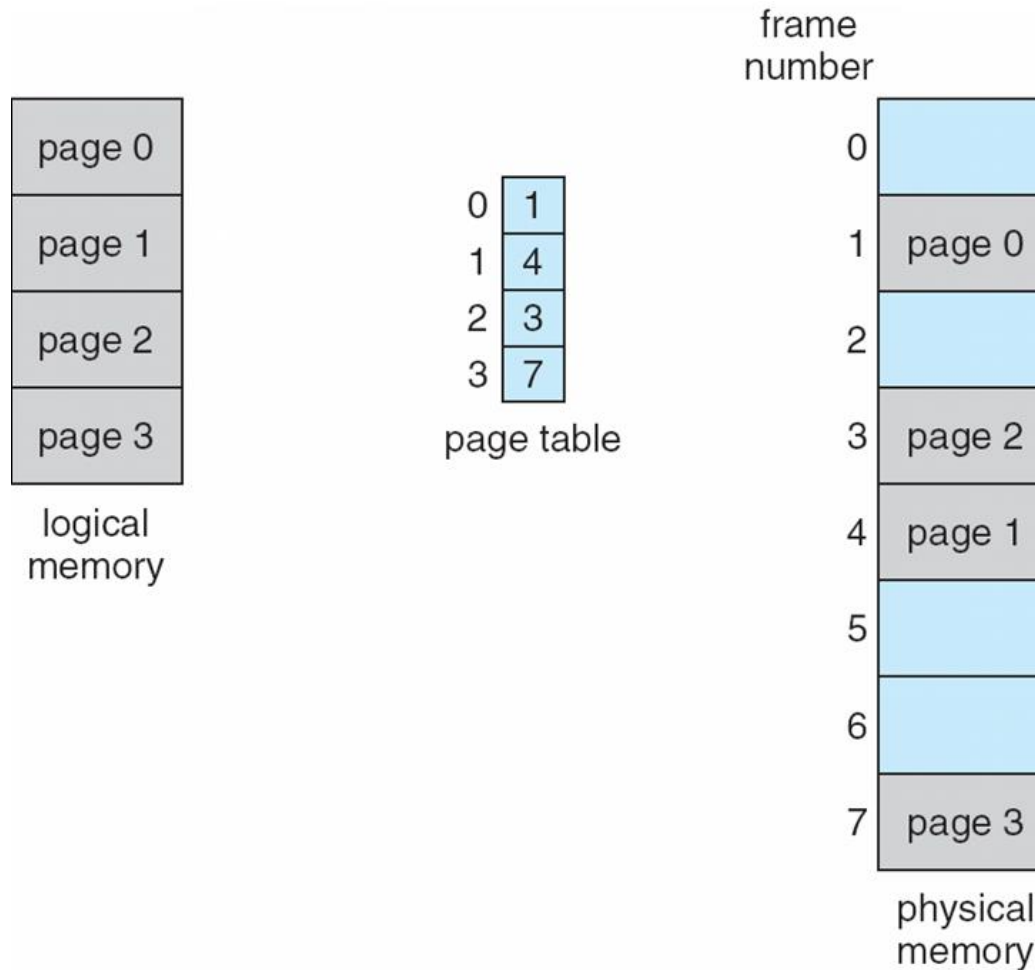
# Paging Hardware







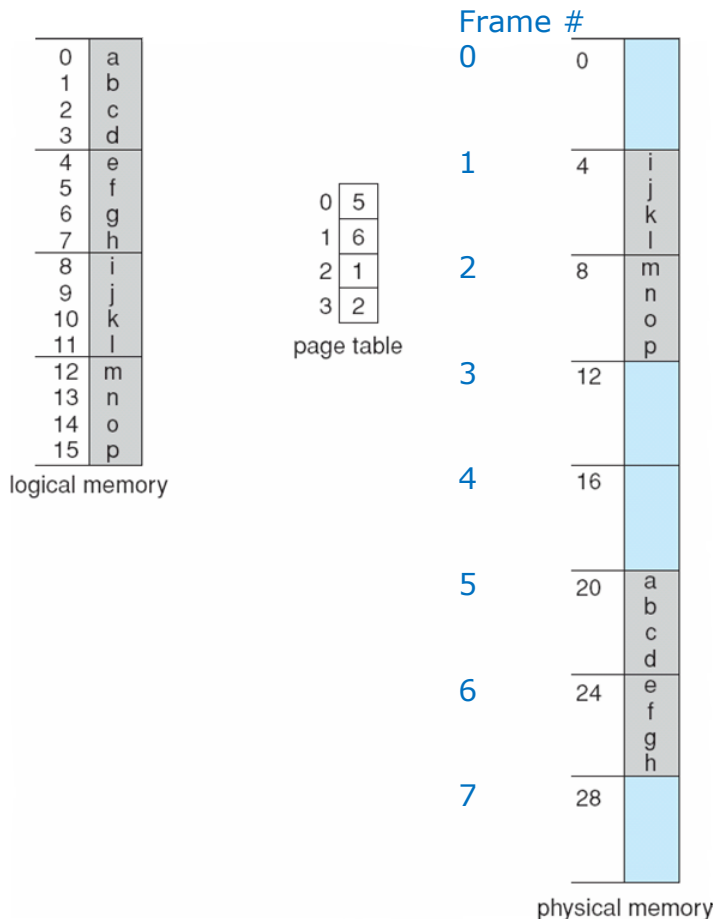
# Paging Model of Logical and Physical Memory





# 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)



Page # range for logical address space:

$$2^{m-n} = 2^2 = 4 \text{ (2 bits)}$$

Total logical address space:

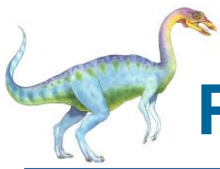
$$2^4 = 16 \text{ bytes (4 bits)}$$

Total physical address space:

$$2^5 = 32 \text{ bytes (8 pages)}$$

( $2^3$  for page#,  $2^2$  for page offset)





# Paging -- Calculating internal fragmentation

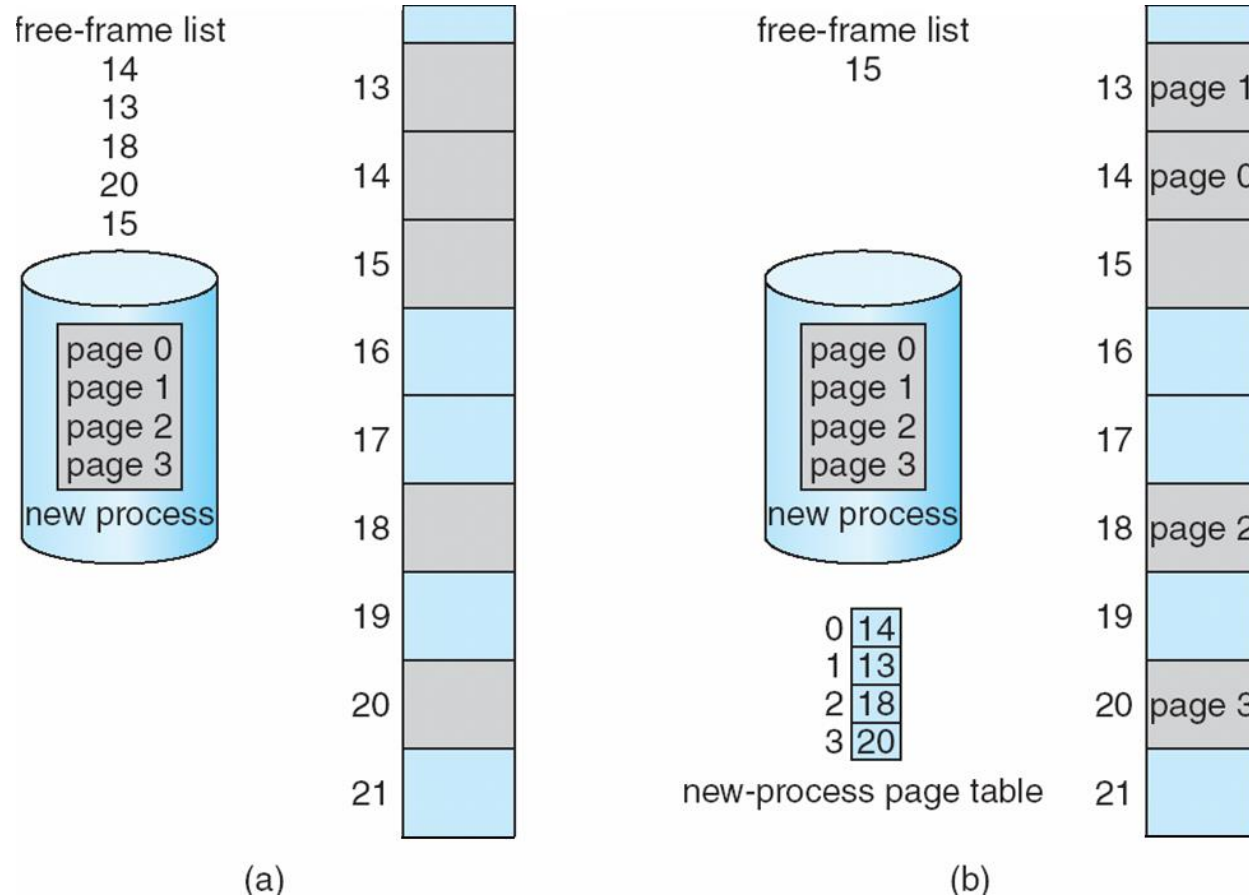
---

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages ( $35 \times 2048 = 71,680$ ) + 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 per-process and requires hardware support
- It is kept in main memory as part of PCB of each process, which is referenced by:
  - **Page-table base register (PTBR)** points to the page table
- Another register is used for protection:
  - **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

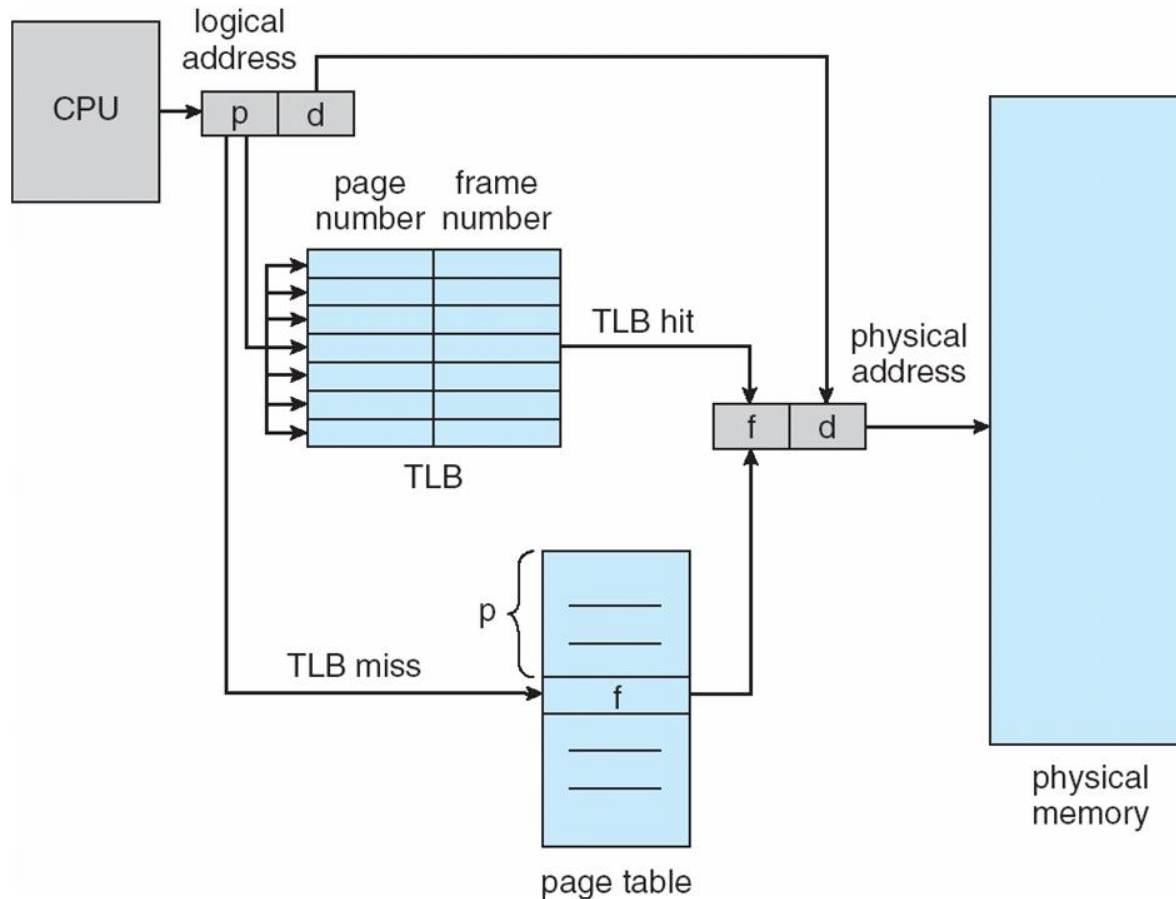
Page #	Frame #

- 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 it takes 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 accesses, 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)**
- Any violations result in a trap to the kernel





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

Assuming 14-bit address space (0-16,383), and a program should only use up to 10,468

Given page size of 2 KB (2,048B,  $6 \times 2,048 = 12,288$ )

Pages 0~5 are mapped normally

Pages 6-7 are invalid

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
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 <i>n</i>

There is a problem with page 5 – it is marked as valid, but the program only uses up to 10,468 (the first byte of page 5), this is an example of internal fragmentation and is wasteful.

Since a process rarely uses all its logical address range, page table could be tailored to the size of actually used range, and hardware support in the form of a **page-table length register** (PTLR) will be checked to verify the valid address range for the process.





# 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

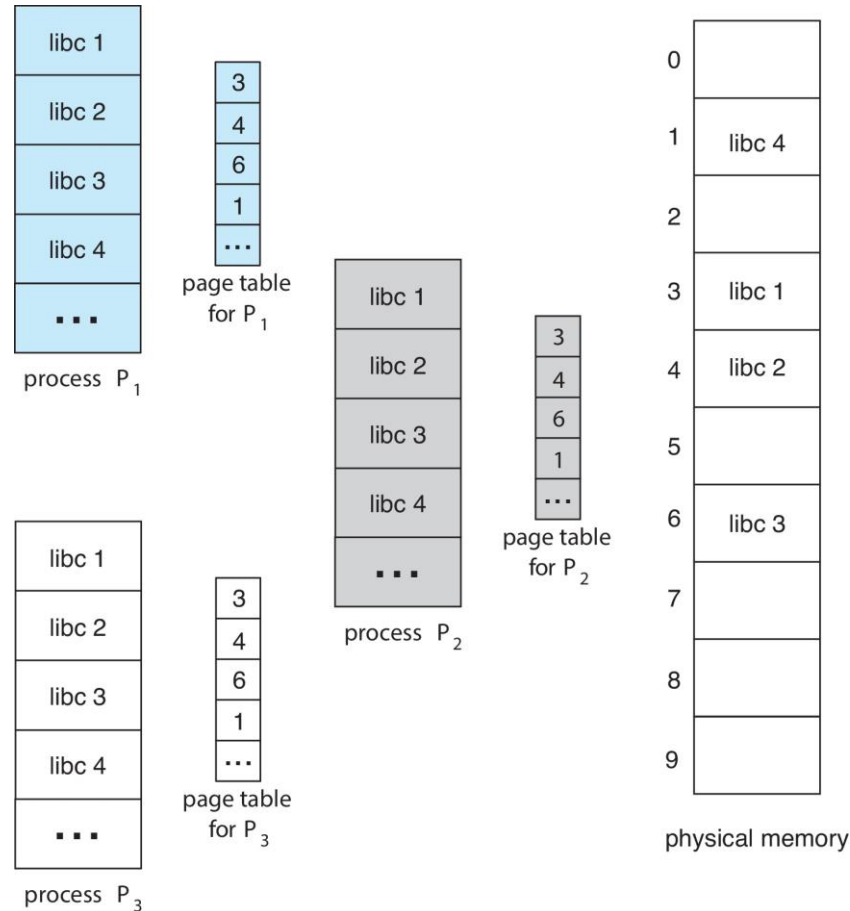
## ■ Example

- The standard C library `libc` provides a portion of the system call interface for many versions of Unix/Linux.
- Since many user processes need it, it can be shared





# Shared Pages Example





# Structure of the Page Table

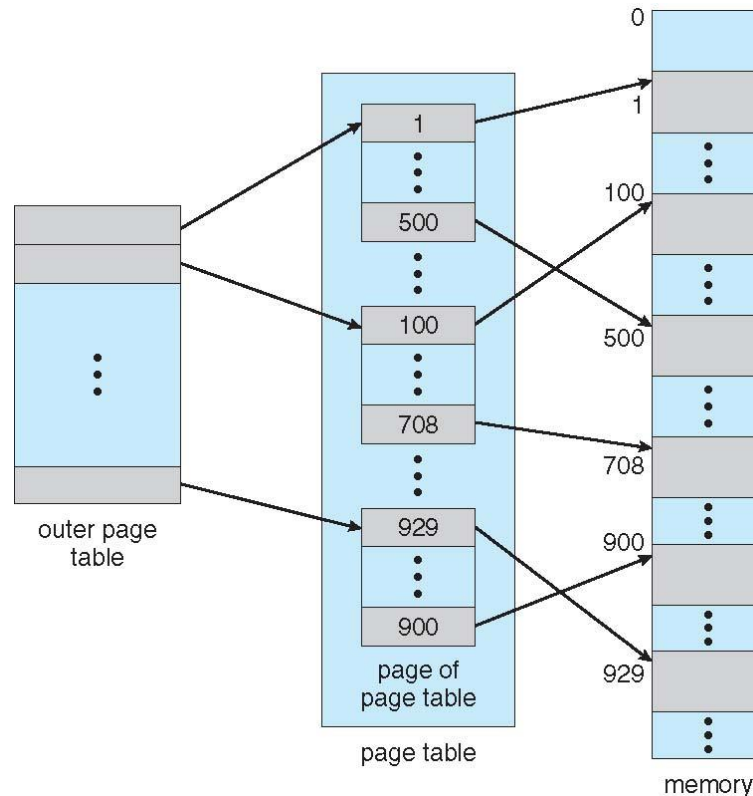
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ( $2^{12}$ )
  - Page table would have 1 million entries ( $2^{32} / 2^{12} = 2^{20}$ )
  - If each entry is 4 bytes → each process needs 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

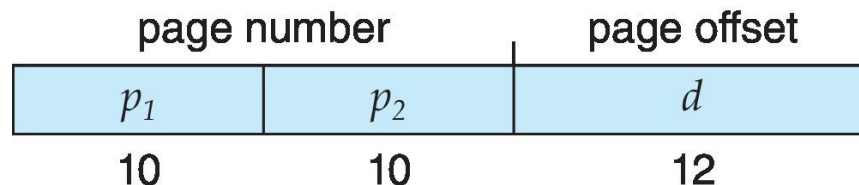
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





# 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:



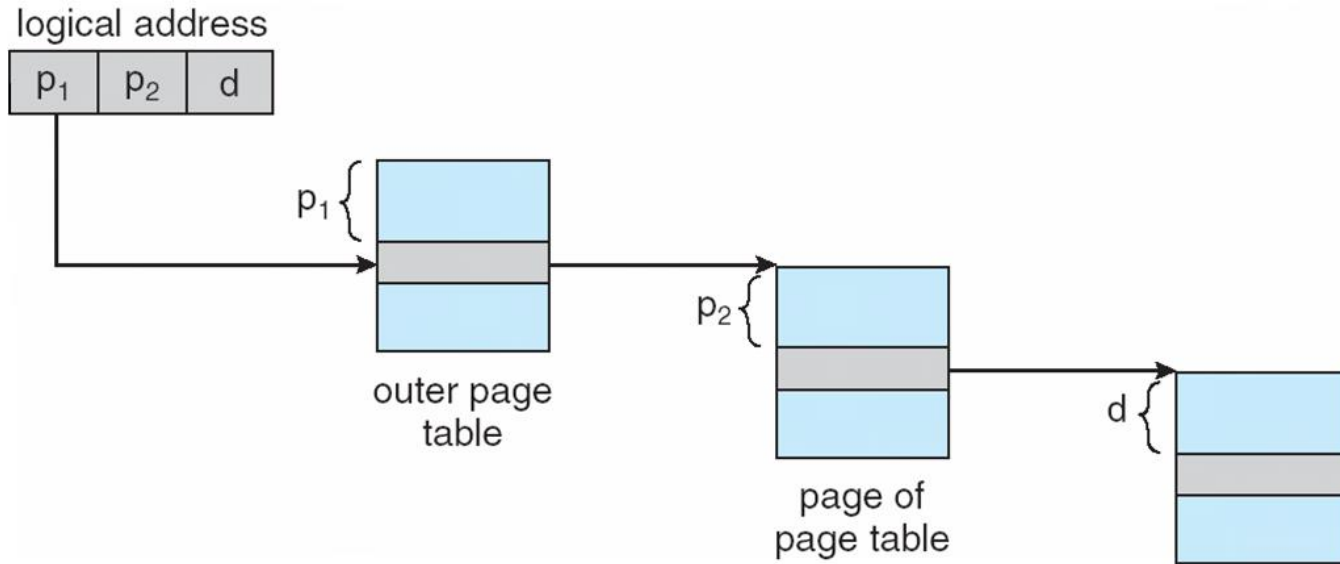
- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table
- Known as **forward-mapped page table**







# Address-Translation Scheme





# 64-bit Logical Address Space

- Even two-level paging scheme is 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	inner page	offset
$p_1$	$p_2$	$d$
42	10	12

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





# Three-level Paging Scheme

outer page	inner page	offset
$p_1$	$p_2$	$d$
42	10	12

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12





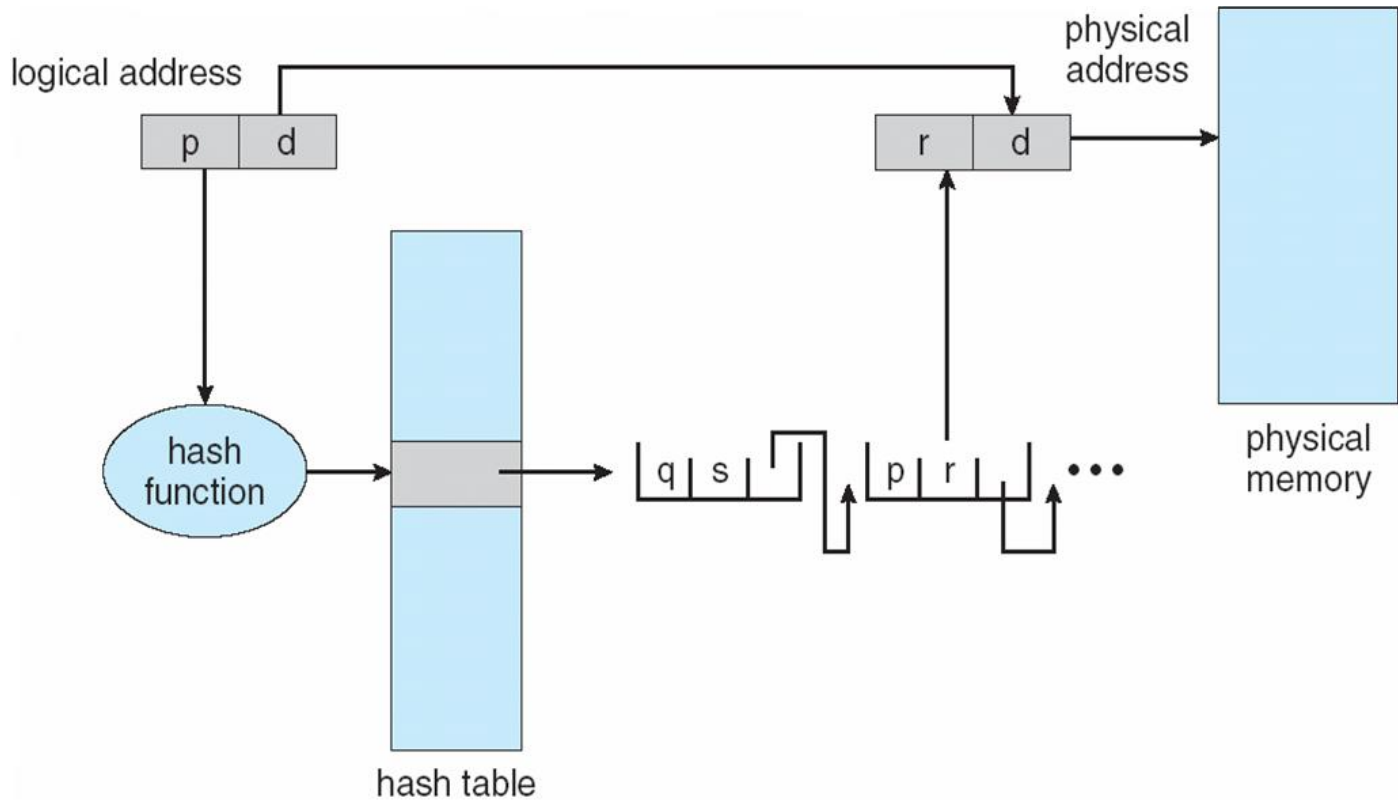
# Hashed Page Tables

- 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
  - Could be considered a tradeoff between linear and hashed page tables
    - ▶ Better than linear table for **sparse** address spaces
    - ▶ Better than hashed table for dense address spaces





# Hashed Page Table





# Inverted Page Table

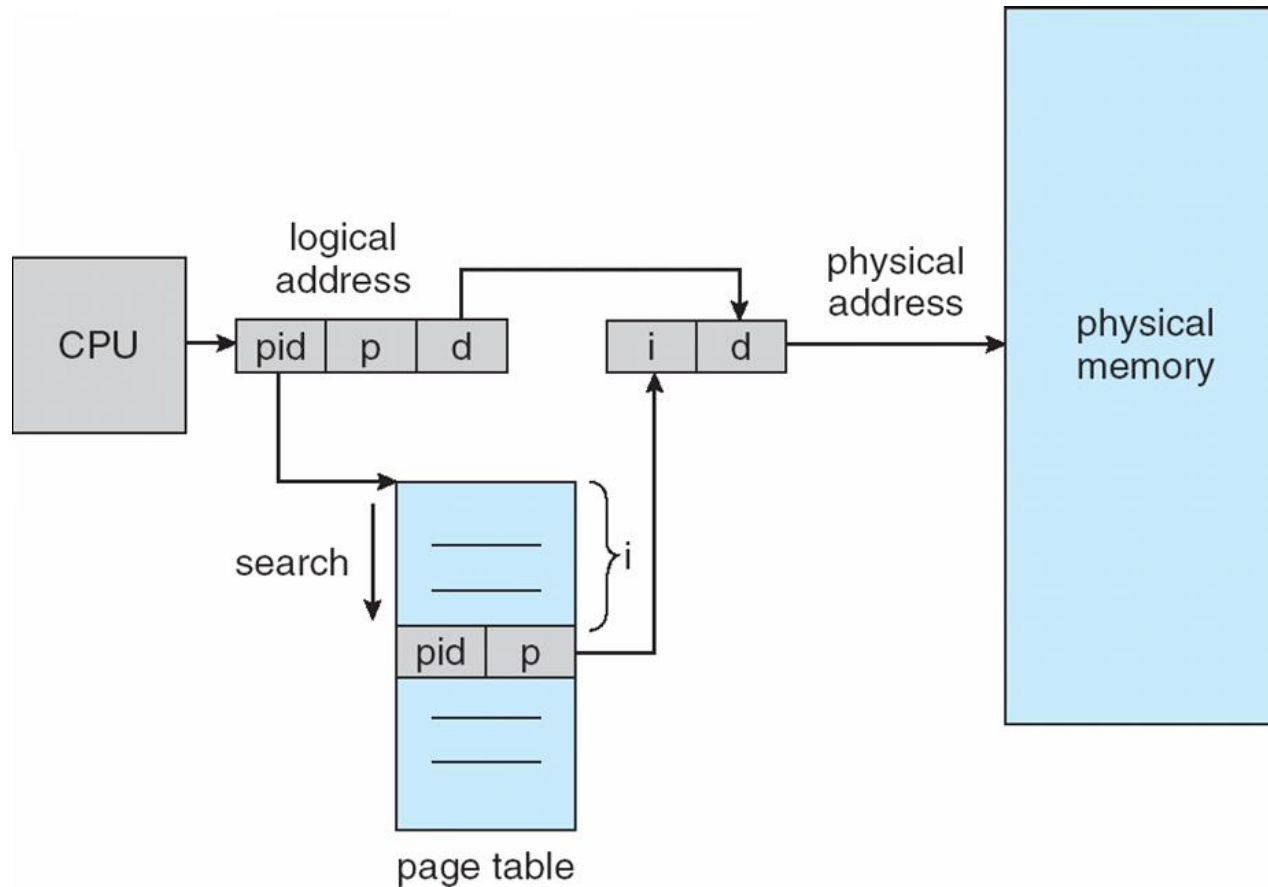
---

- 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? It can't.
  - One mapping of a virtual address to one physical address
  - A reference by another process sharing the memory results a page fault and replace the mapping with a different virtual address





# Inverted Page Table Architecture



Only one page table in the system





# 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 existing 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
- Types of Swapping
  - **Standard Swapping**: moving entire processes between main memory and a backing store. Not used in modern OS
  - **Swapping with Paging**: used by Linux and Windows and commonly called *paging*, so *swapping* now refers to standard swapping
  - **Swapping on Mobile Systems**: mobile systems typically don't support swapping in any form due to hardware limitation

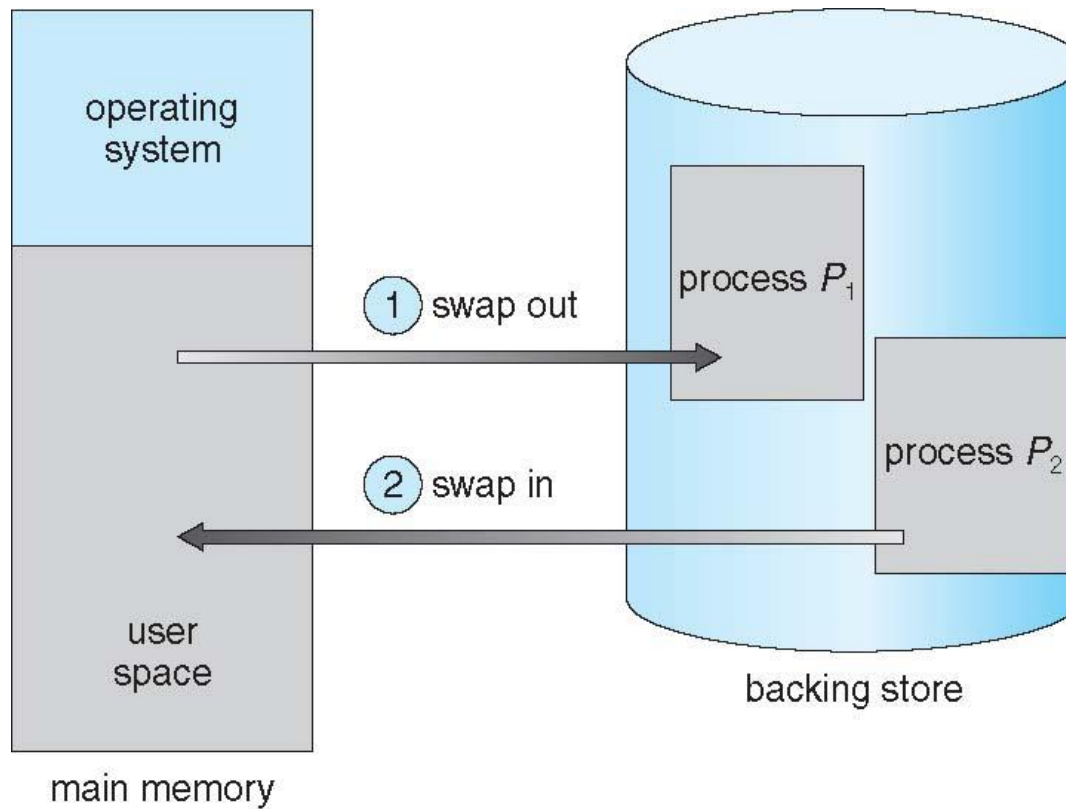






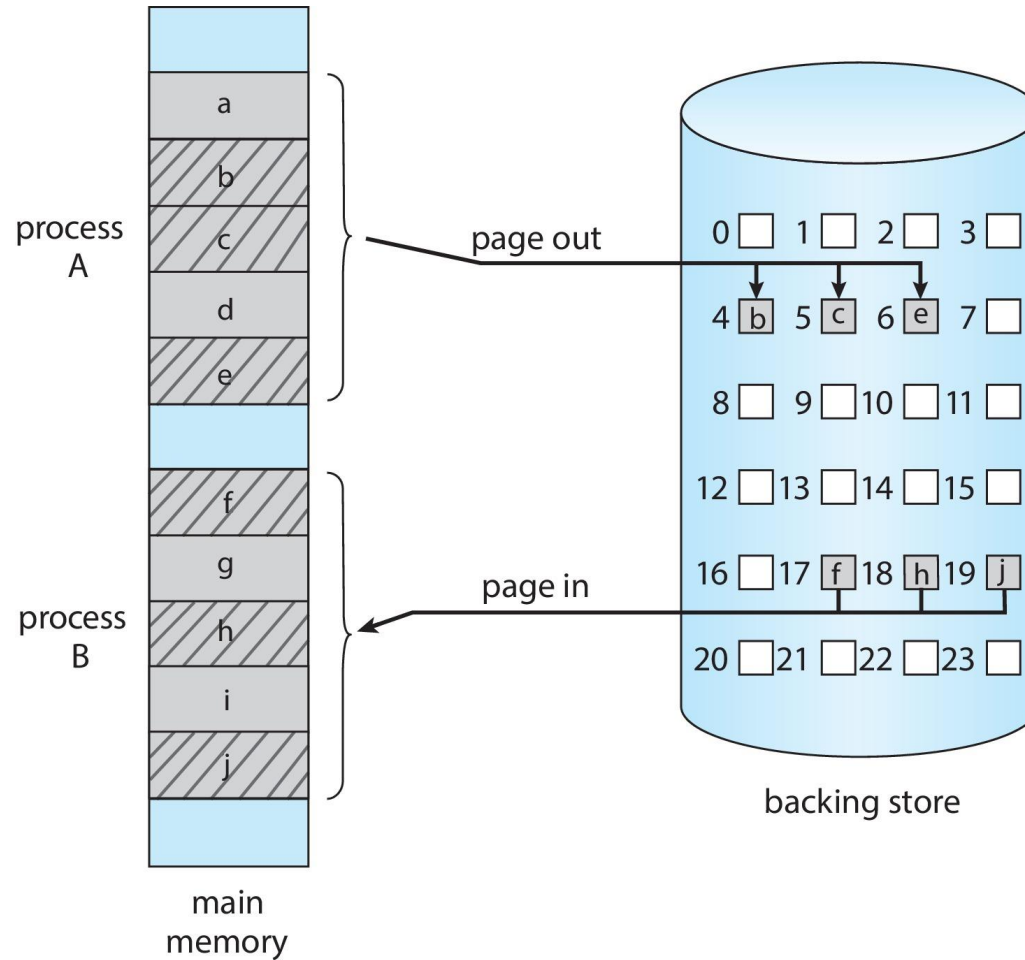
# Schematic View of Swapping

## Standard Swapping





# Swapping with Paging

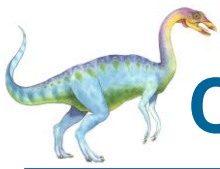




# Context Switch Time including Swapping

- Swapping generally indicates a shortage of physical memory
- 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
- A 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 4000 ms (4 seconds)
- Can 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 free memory is low, but first writes **application state** to flash for fast restart
  - Developers for mobile systems must carefully allocate and release memory to ensure their apps don't use too much memory or cause memory leaks





# 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 x86-64 architecture
- Many variations in the chips, cover the main ideas here





# Example: The Intel IA-32 Architecture

---

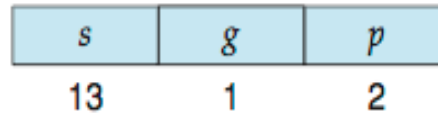
- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB (32-bit)
  - Up to 16 K segments ( $2^{14}$ ) per process
  - Divided into two partitions
    - ▶ First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
    - ▶ Second 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 (48-bit)
  - Selector given to segmentation unit
    - ▶ Which produces linear addresses  
s: segment, g: local/global, p: protection



+ 32-bit offset with the segment

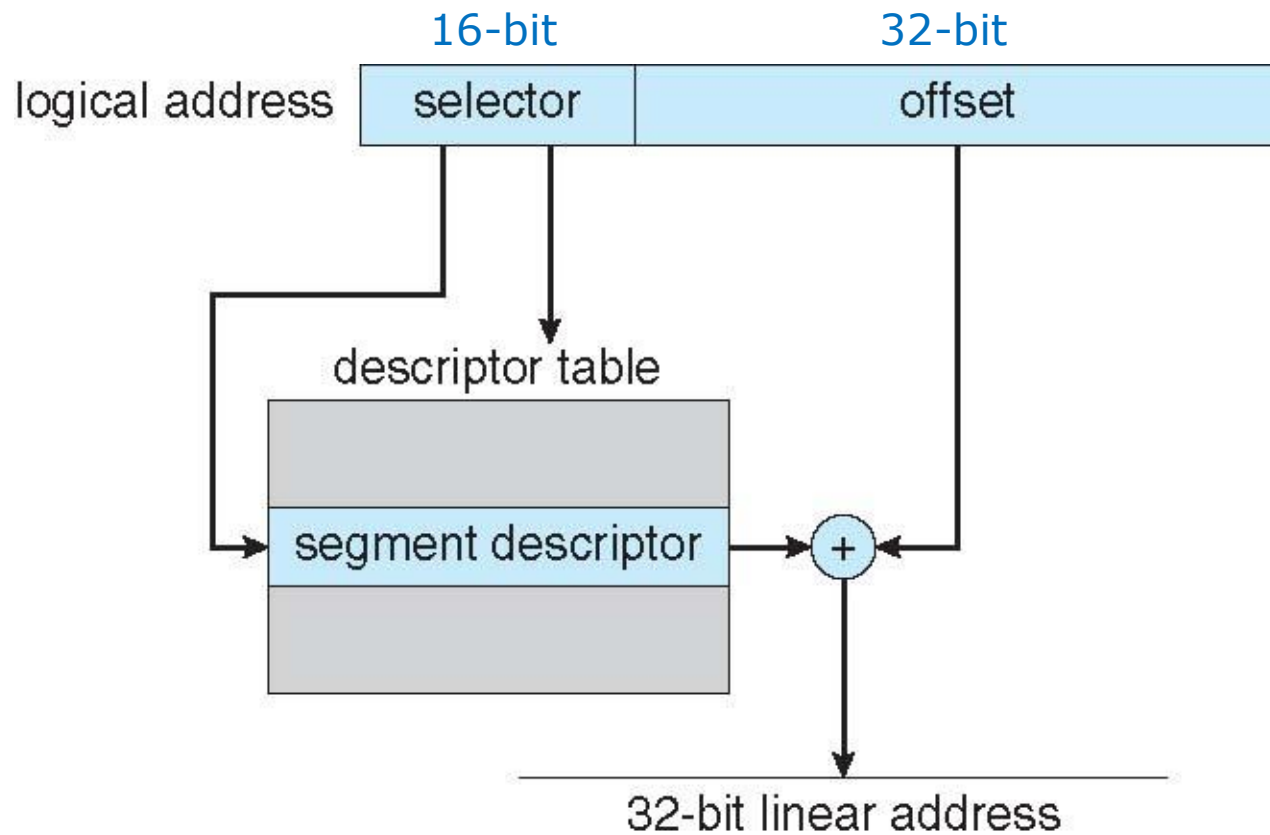
- Linear address given to paging unit
  - ▶ Which generates physical address in main memory
  - ▶ Paging units form equivalent of MMU
  - ▶ Pages sizes can be 4 KB or 4 MB

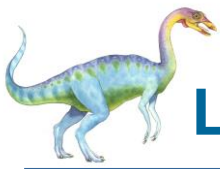




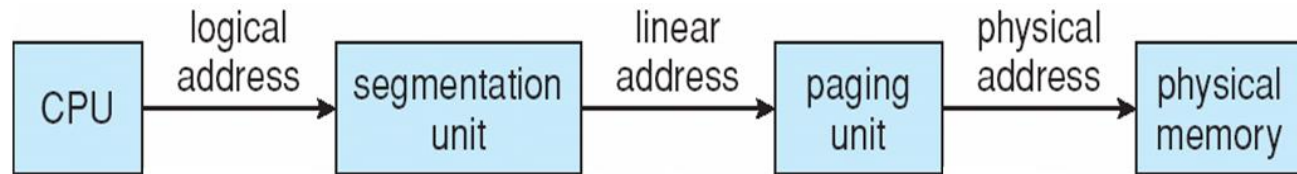


# Intel IA-32 Segmentation

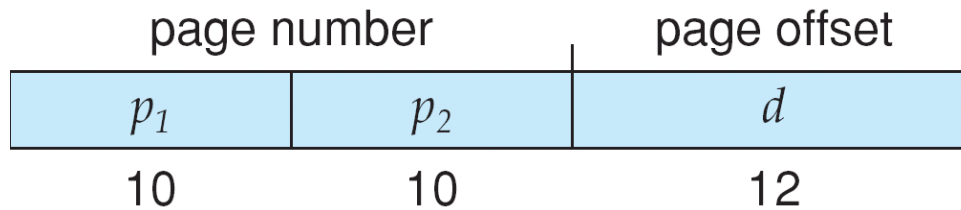




# Logical to Physical Address Translation in IA-32



32-bit linear address

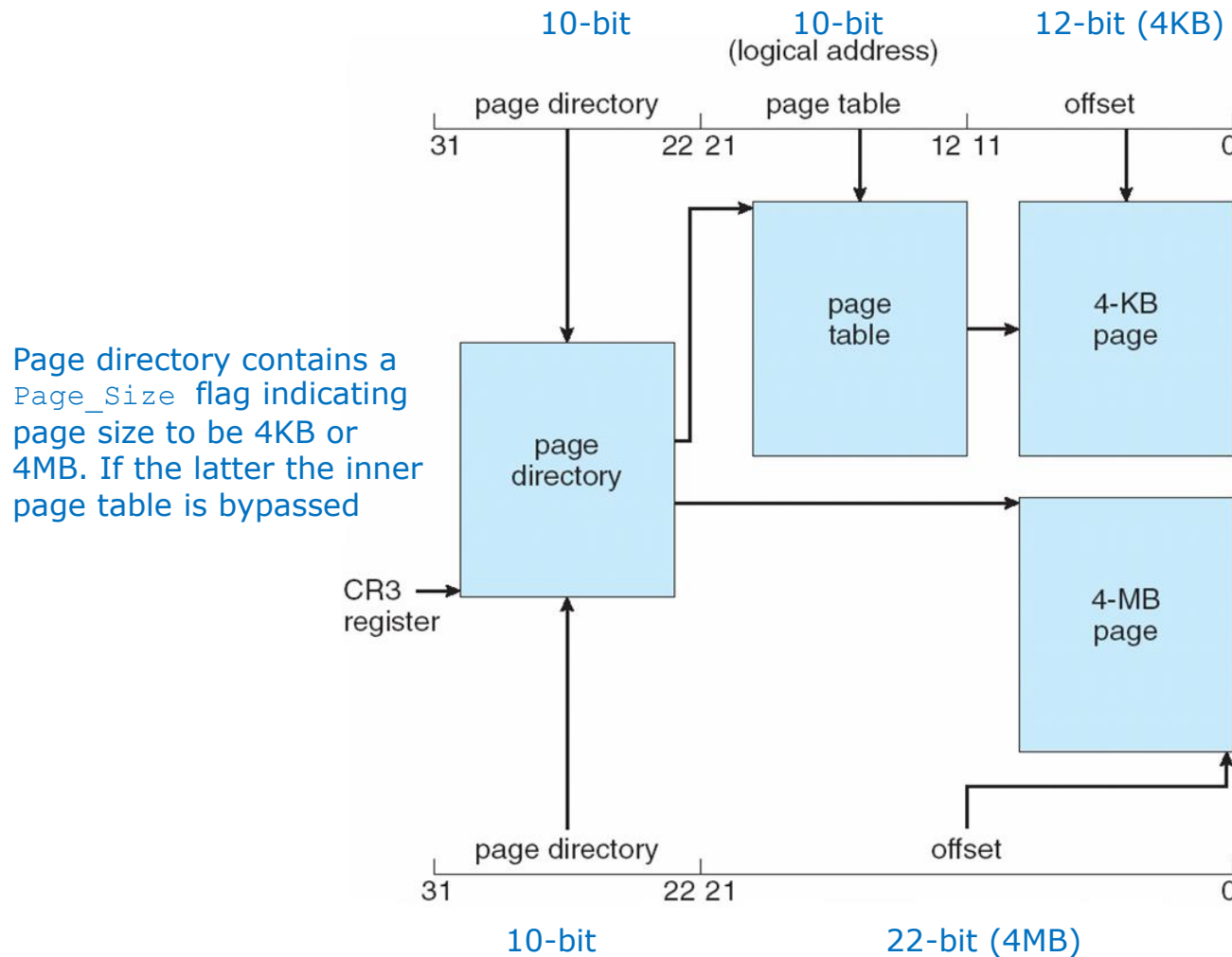


$2^{12} = 4\text{KB}$  pages





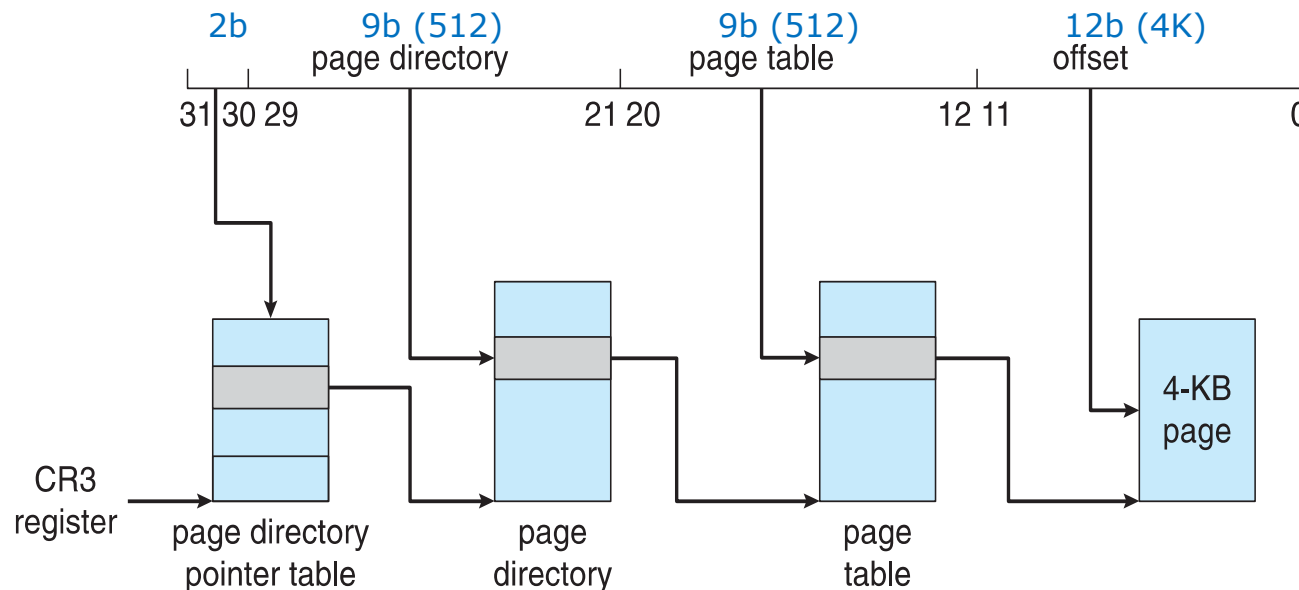
# Intel IA-32 Paging Architecture





# Intel IA-32 Page Address Extensions

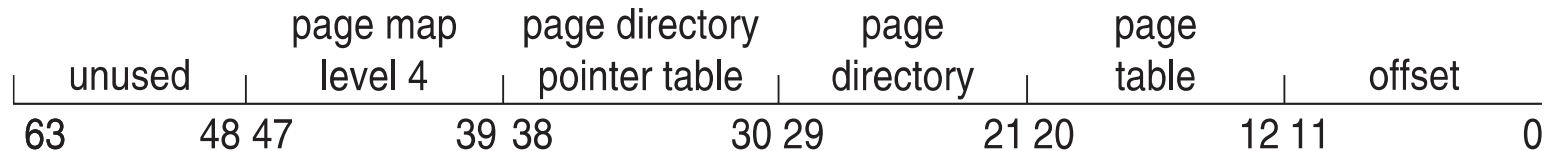
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
    - Only 24-bit were used for the base address (compared to 20-bit w/o extensions)
  - Net effect is increasing address space to 36 bits – 64GB of physical memory (only Linux/MacOS support it, not 32-bit Windows)





# Intel x86-64

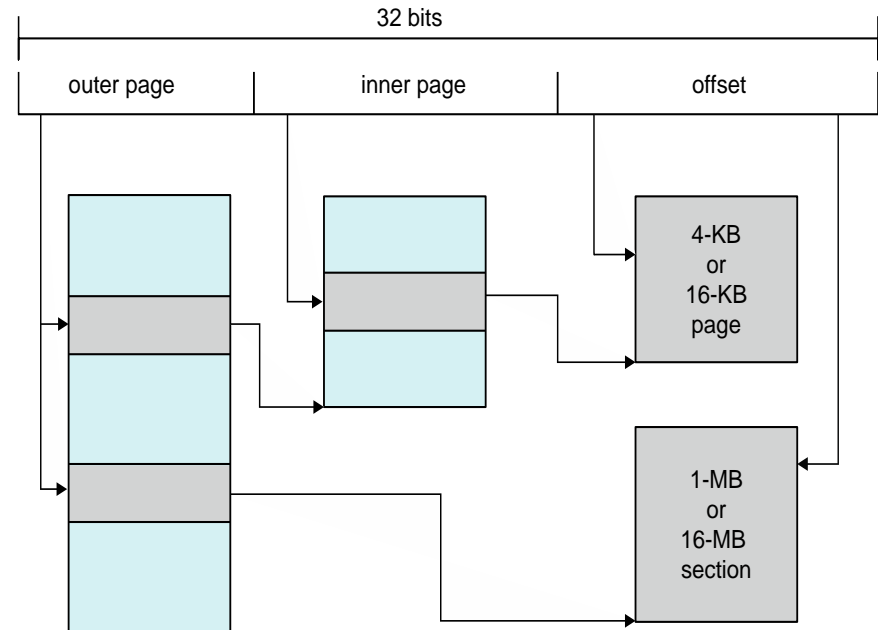
- Current generation Intel x86 architecture
- 64 bits is ginormous ( $> 16$  exabytes) (  $1 \text{ EB} = 1\text{K PB} = 1\text{M TB}$  )
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 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 for outer and inner levels. If both miss, a page table walk must be performed by CPU



# End of Chapter 9

---

