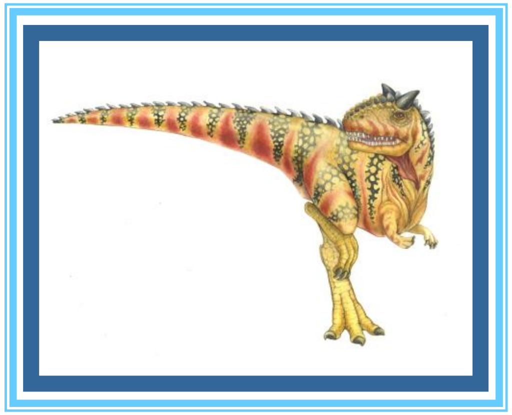
# Chapter 9:  Main Memory



# 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

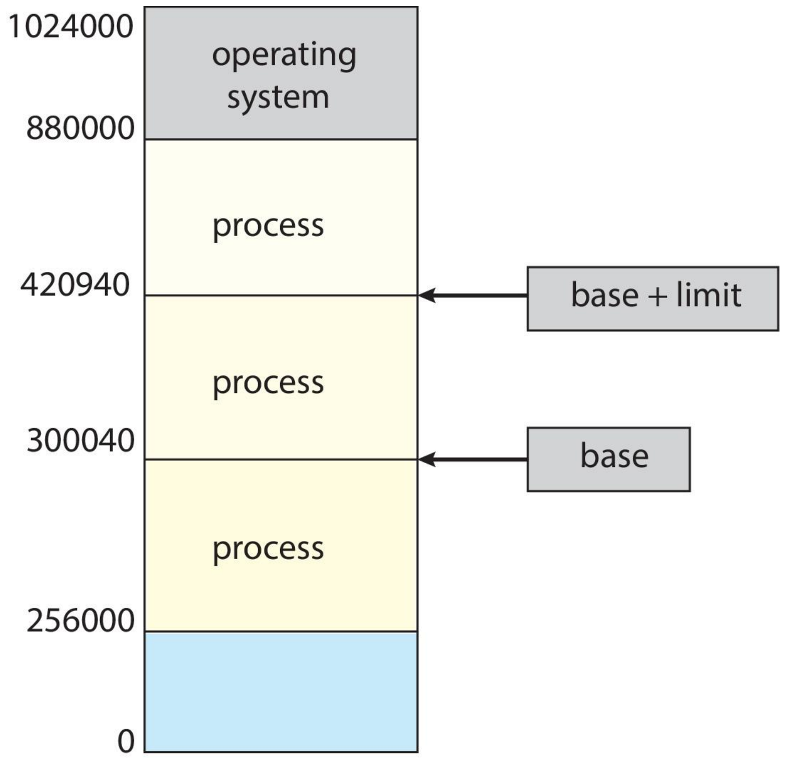
- Program must be brought (from disk) into memory and placed within a process for it to be run
- The CPU fetches instructions from memory according to the PC.
  - Fetched instructions may cause additional loading from and storing to specific memory addresses.
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data + write requests
  - does not know how addresses are being generated (e.g., instruction counter, indexing, indirection, etc)
  - does not know the address is for instructions or data

## Background

- Main memory and registers are only storage CPU can access directly
  - If the desired data are not in memory, they must be moved there before the CPU can operate on them
- Register are built into each CPU core.
  - Accessible within one cycle of the CPU clock.
- Main memory can take many cycles; processor needs to **stall**
- **Cache** sits between main memory and CPU registers

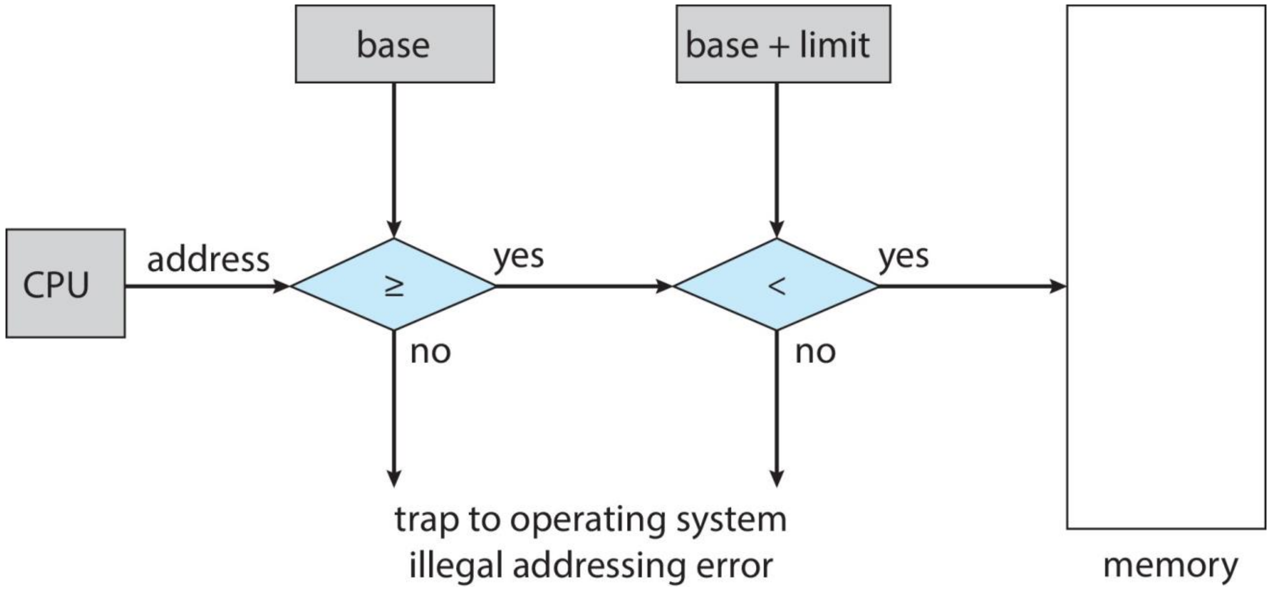
# Protection

- Along with relative speed of memory access correct operation must be ensured.
- Protection of memory required to ensure correct operation
- Need to ensure that a process can access only those addresses in its address space.
  - Each process should have separate memory 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

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the base and limit registers.



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

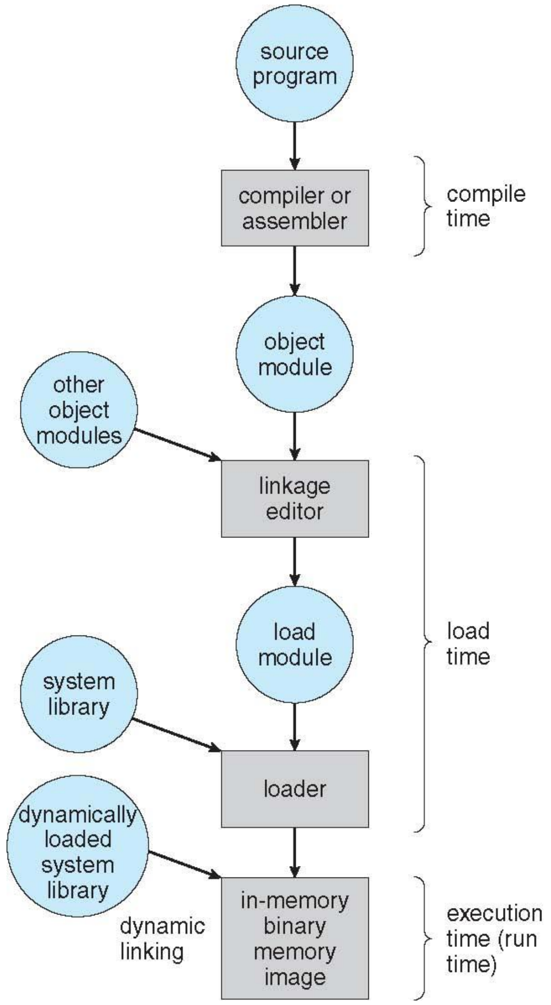
## Address Binding

- Addresses 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 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)

## Multistep Processing of a User Program

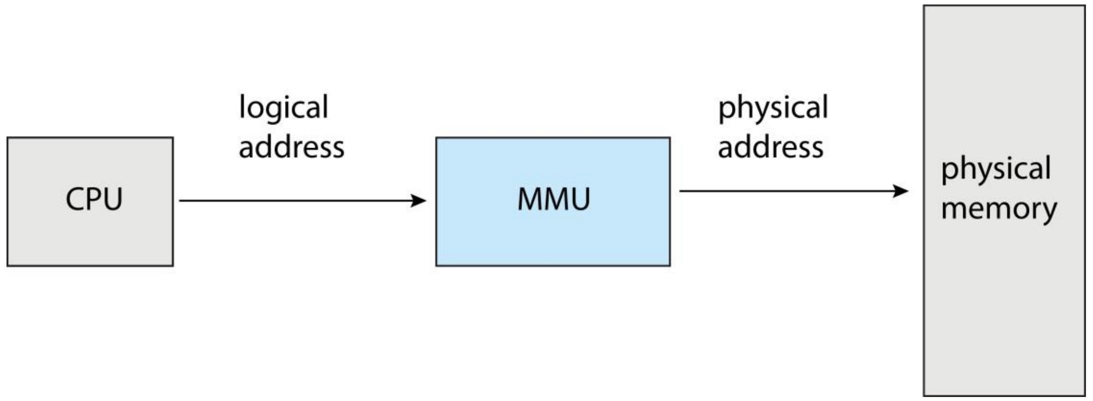


## Logical vs. Physical Address Space

- 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)

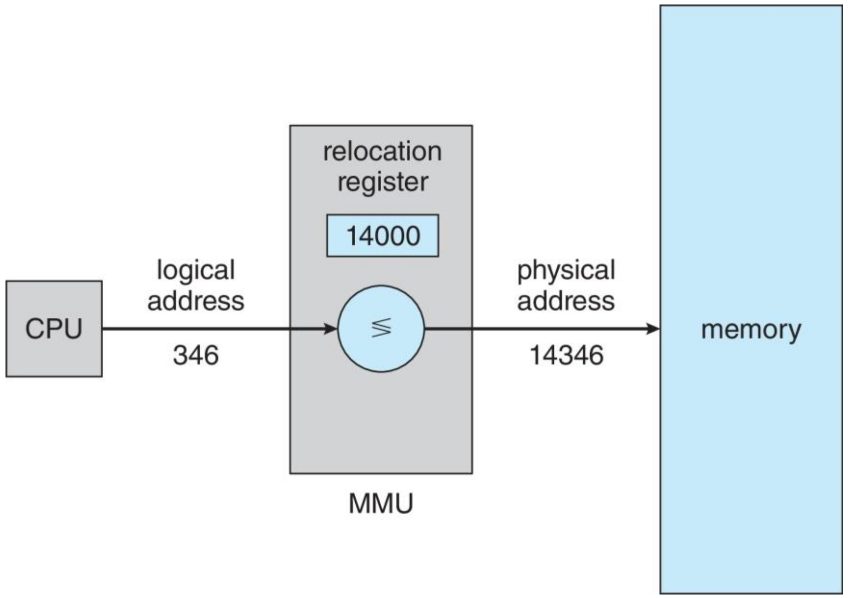
- Hardware device that at run time maps virtual to physical address



- 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



## Dynamic Loading

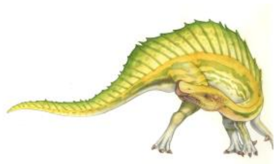
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- When a routine needs to call another routine, the calling routine first checks the availability of other routine in the memory
  - If not, relocatable linking loader is called to load the desired routine and to update the program's address tables to reflect this change.
- No special support from the operating system is required
  - Implemented through program design (How??)
  - OS may help by providing libraries to implement dynamic loading

# Dynamic Linking

- **Static linking**
  - System library modules get fully included in executable modules
  - wasting both disk space and main memory usage
- **Dynamic linking** –linking postponed until execution time
- Small piece of code (i.e., **stub)**, used as reference to the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for system libraries
- Dynamically linked libraries can be shared among multiple processes, thus, also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

## Contiguous Allocation

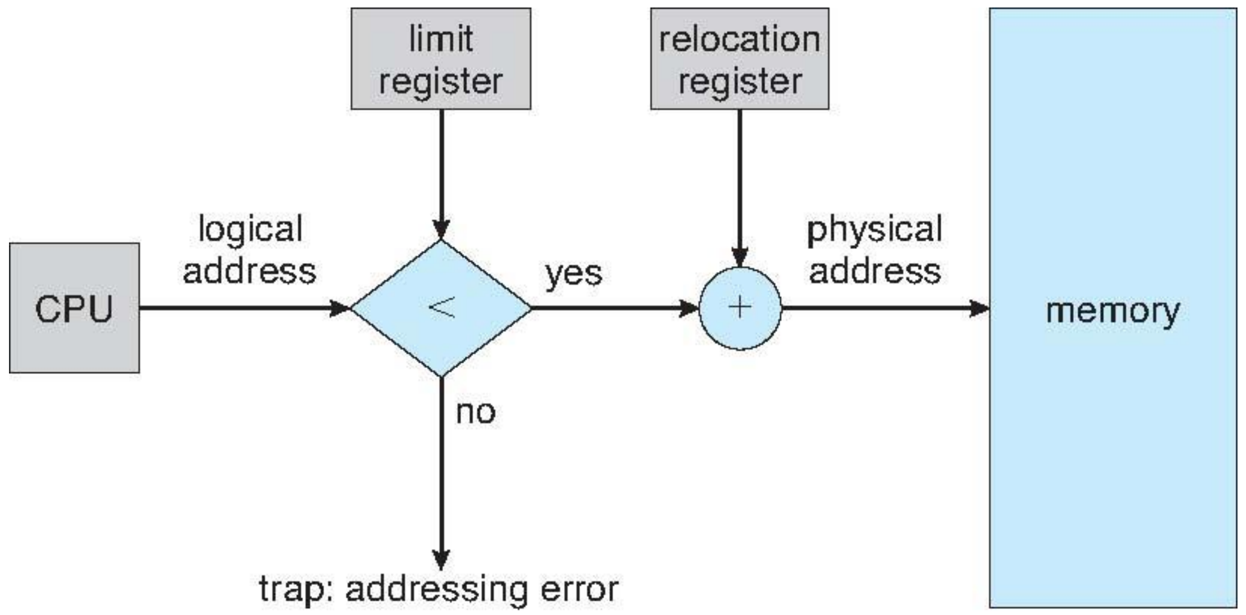
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
- We can place the operating system in either low memory addresses or high memory addresses. Linux and Windows place the operating system in high memory
  - User processes then held in low/high memory accordingly
  - Each process contained in single contiguous section of memory



## Contiguous Allocation (Cont.)

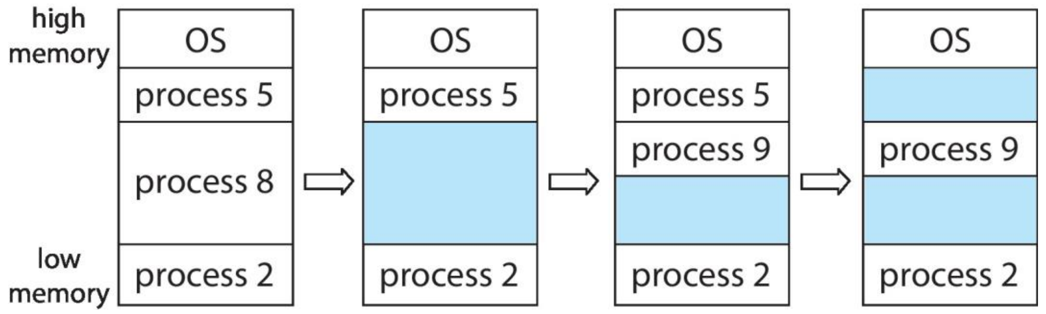
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically* by adding the value in the relocation register
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values.

## Hardware Support for Relocation and Limit Registers



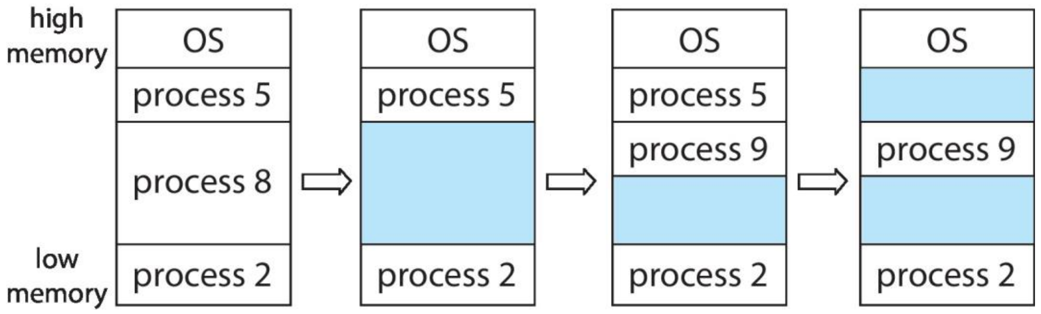
## Variable Partition

- OS keeps a table indicating which parts of memory are available and which are occupied.
    - Initially, one large block (single hole) of memory is available for user processes.



# 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
  - Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.
- **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

Based on simulation, first-fit and best-fit can be considered 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
- First fit statistical analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - **50-percent rule**

The "50-percent rule" in memory allocation and fragmentation states that when using the first-fit algorithm for allocating memory blocks, you can expect that approximately 50% of the allocated blocks will be lost to fragmentation. This fragmentation occurs because of the way memory is allocated and freed, leading to small unusable gaps between allocated blocks.

In statistical analysis, if you have N blocks allocated, the rule suggests that about 0.5N blocks are lost due to fragmentation.

## 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

involves moving allocated blocks together to consolidate free memory into larger contiguous blocks. It can be time-consuming and may require pausing processes, but it effectively reduces fragmentation.

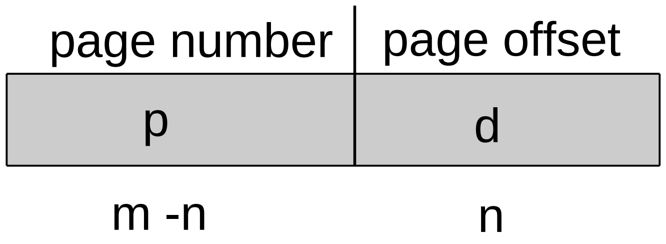
In a paging system, memory is divided into fixed-size pages. This eliminates the need for contiguous allocation and can help avoid external fragmentation since processes are allocated memory in pages rather than contiguous blocks.

# Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation and the associated need for compaction
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide programs logical memory into blocks of same size called **pages**
- Keep track of all free frames
  - Any page ( from any process ) can be placed into any available frame.
- Set up a **page table** to translate logical to physical addresses
- Still have Internal fragmentation

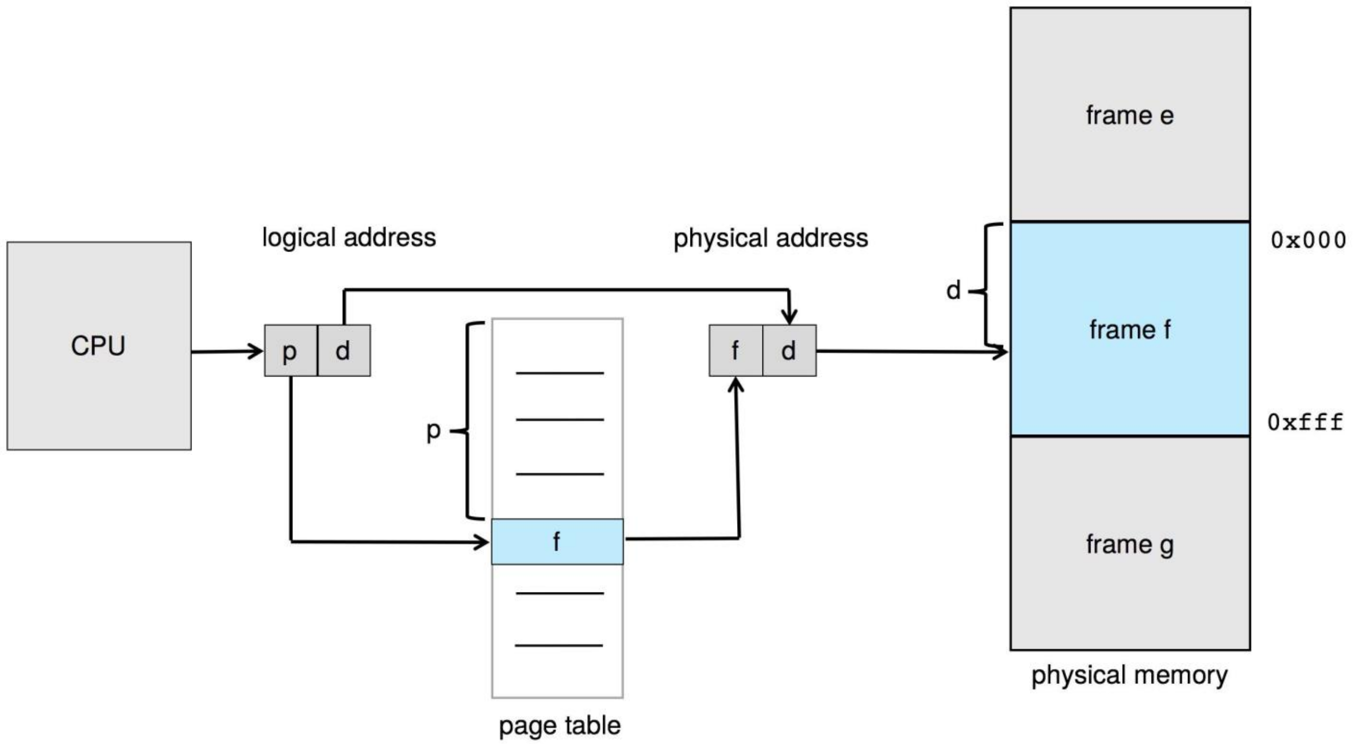
# Address Translation Scheme

- 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



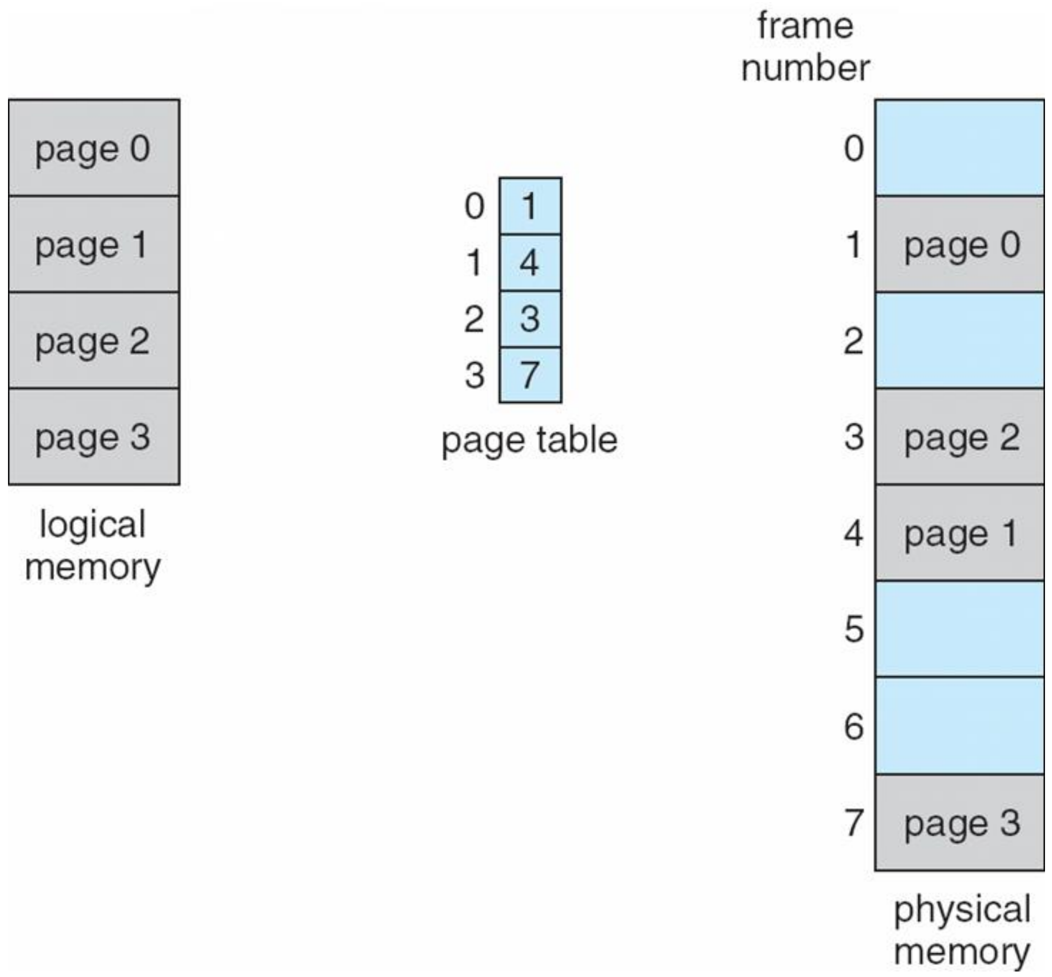
  - For given logical address space $2^m$ and page size $2^n$

# Paging Hardware



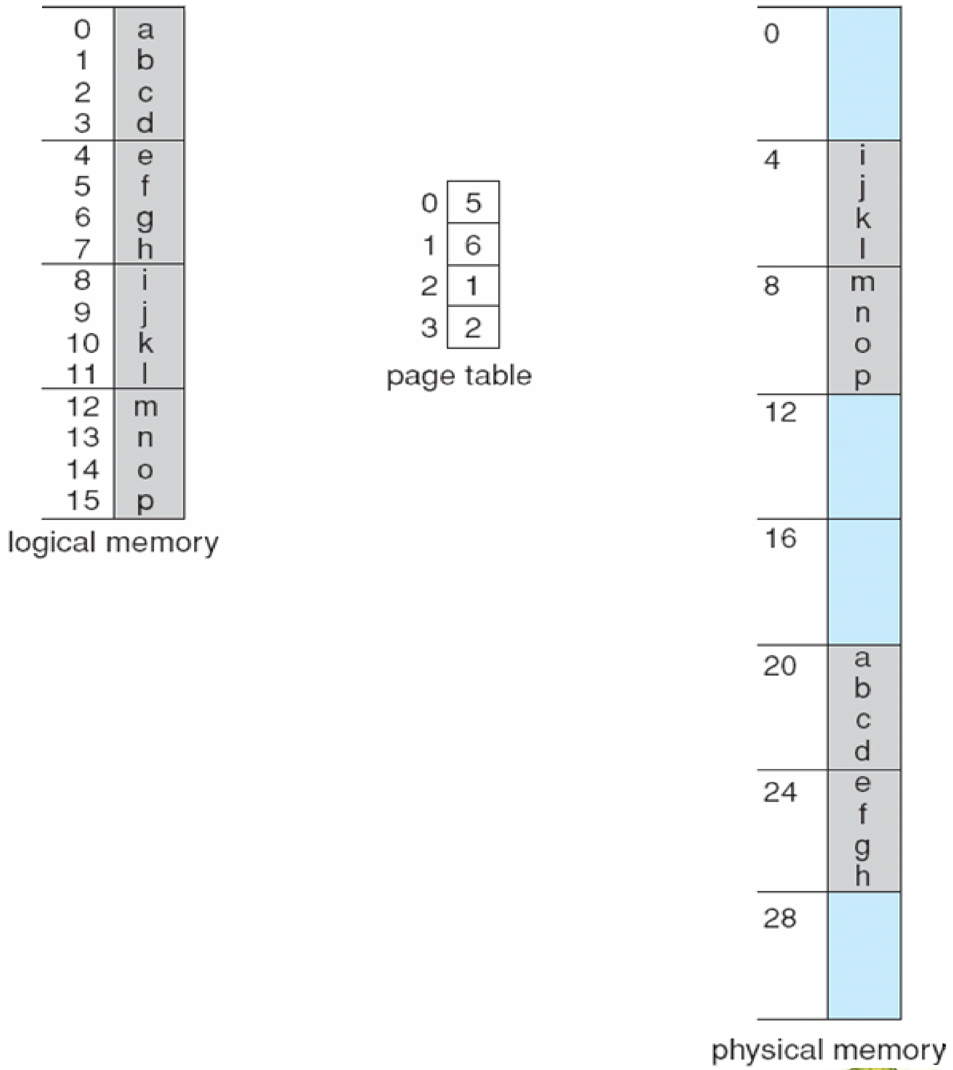
Note: Pages and frames are of same size.

# Paging Model of Logical and  Physical Memory



# Paging Example

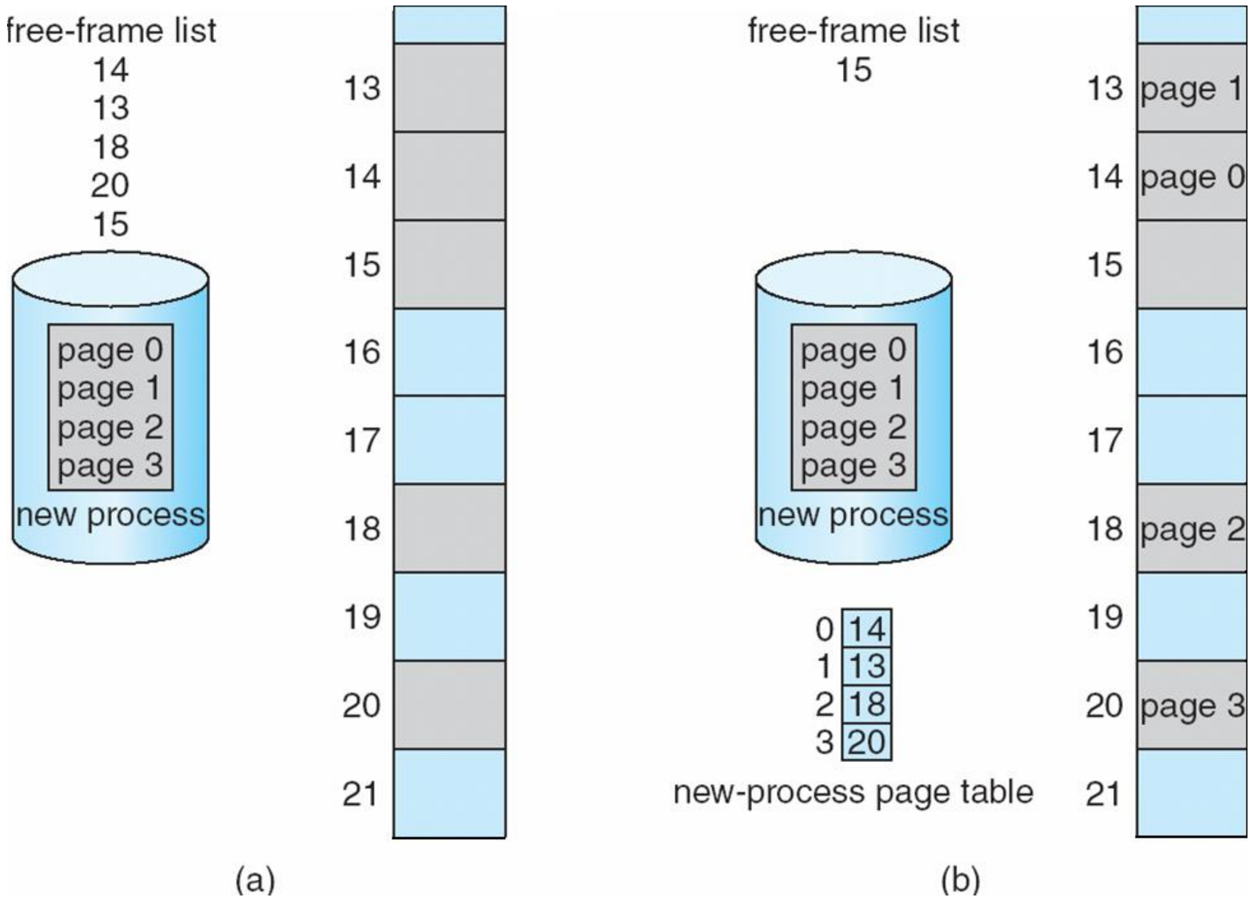
- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory.



# 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
  - Inefficient from the perspective of disk I/O
- With the advancement in processor page sizes are growing
  - On x86-64 systems, Windows 10 supports page sizes of 4 KB and 2 MB
- For Linux default page size (typically 4 KB) and an architecture dependent larger page size
  - System call getpagesize() or command getconf PAGESIZE

## 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**).
  - When the associative memory is presented with an item, the item is compared with all keys simultaneously.

# Translation Look-Aside Buffer

- The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

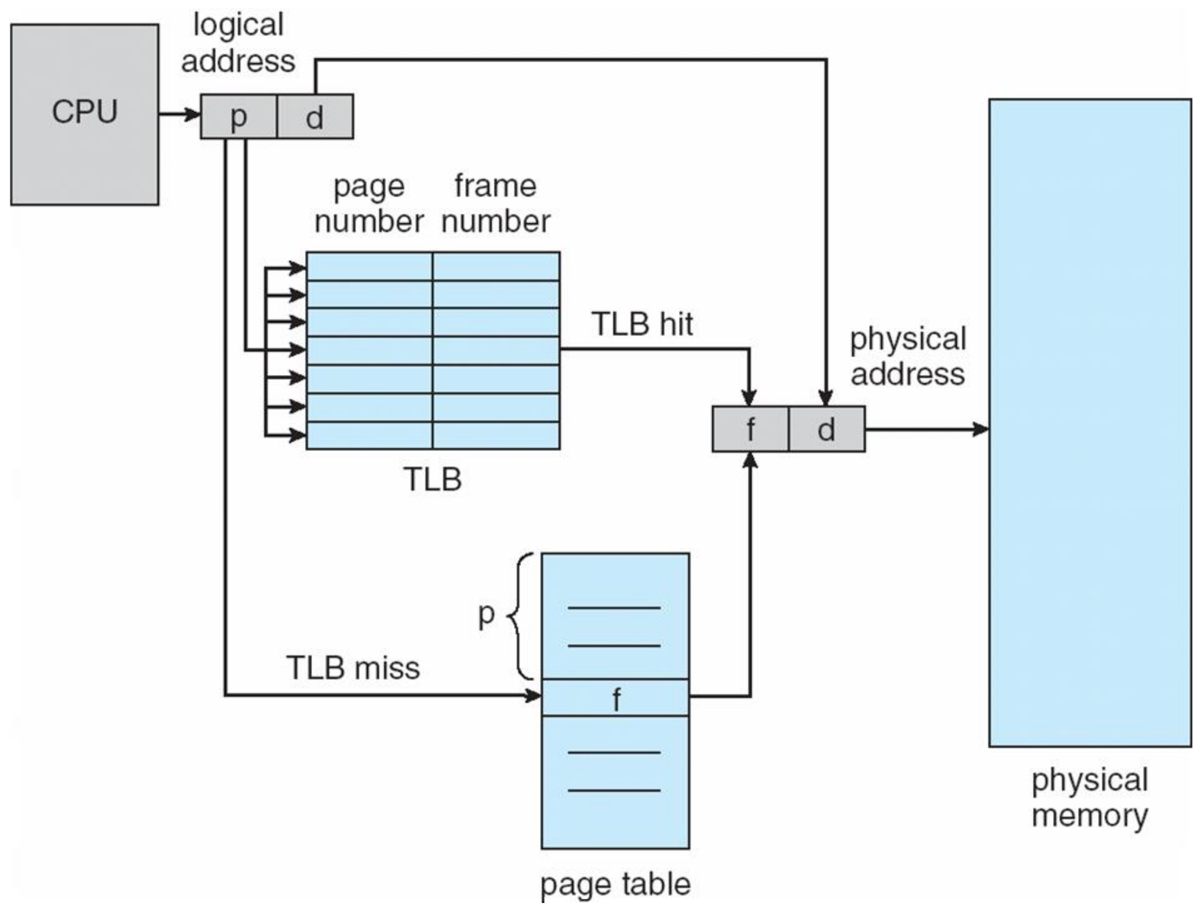
# Hardware

- 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**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

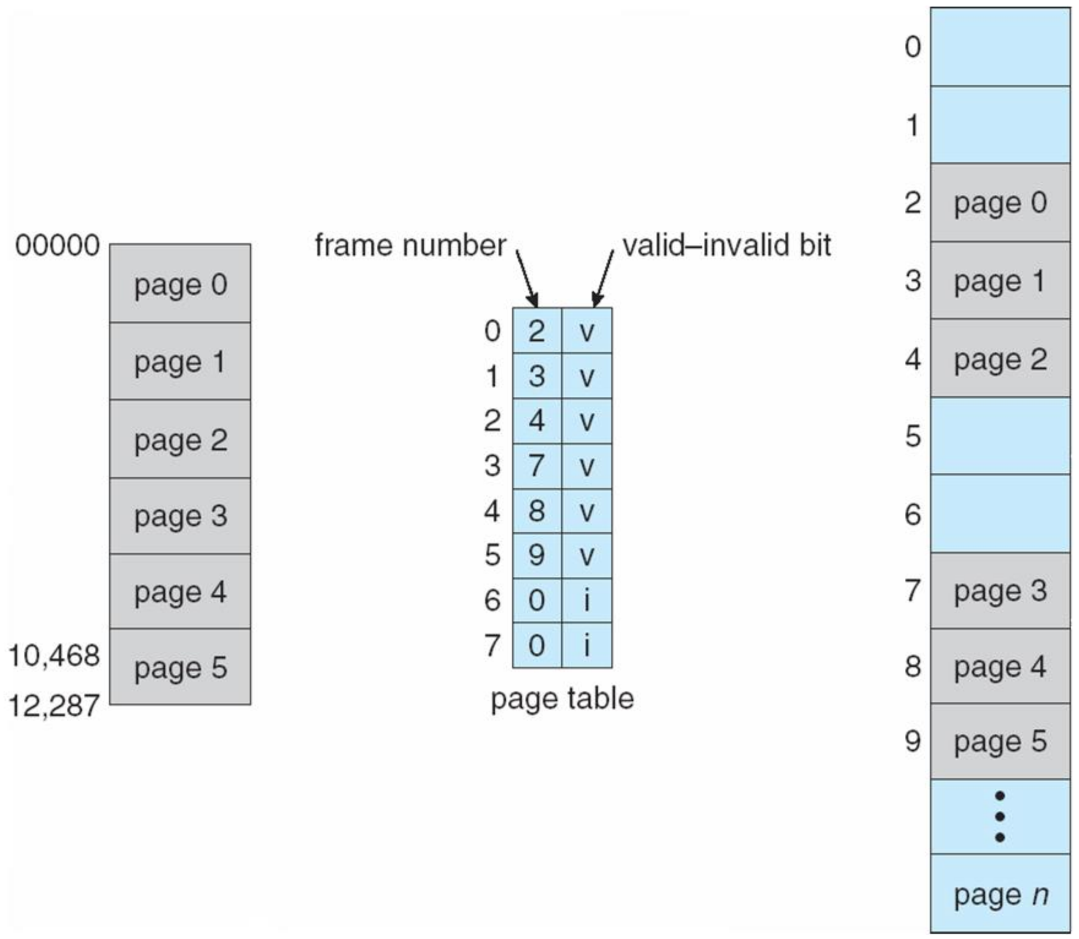
  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  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

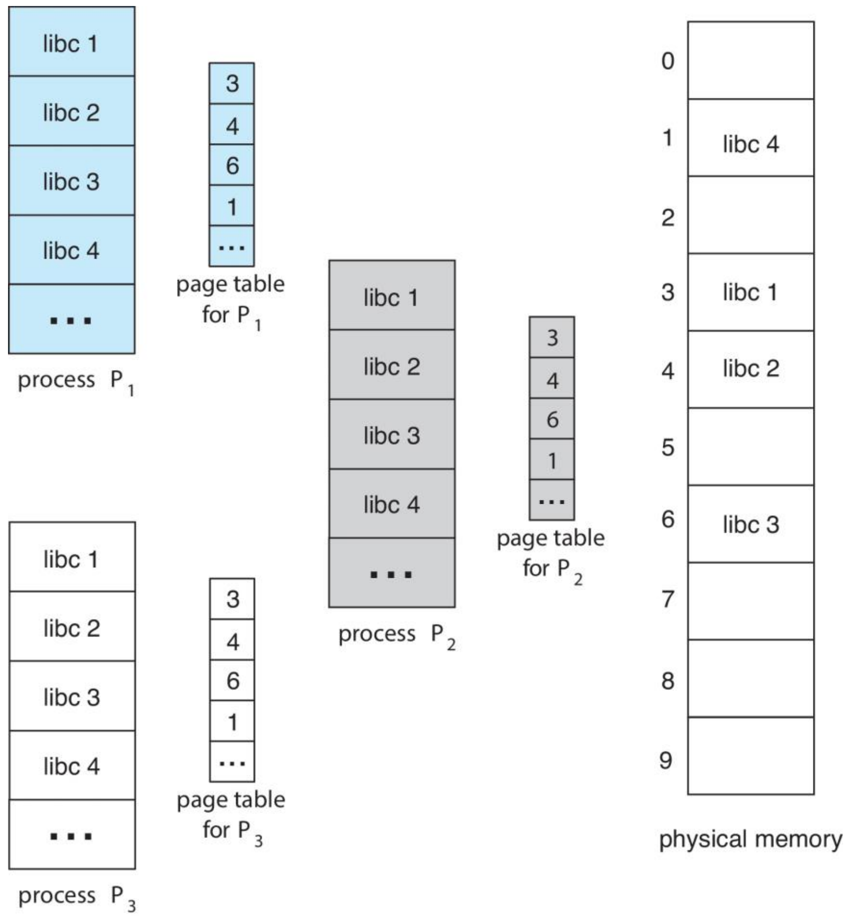
## Valid (v) or Invalid (i) Bit In A Page Table



# Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
    - Reentrant code is non-self-modifying code: it never changes during execution
  - 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

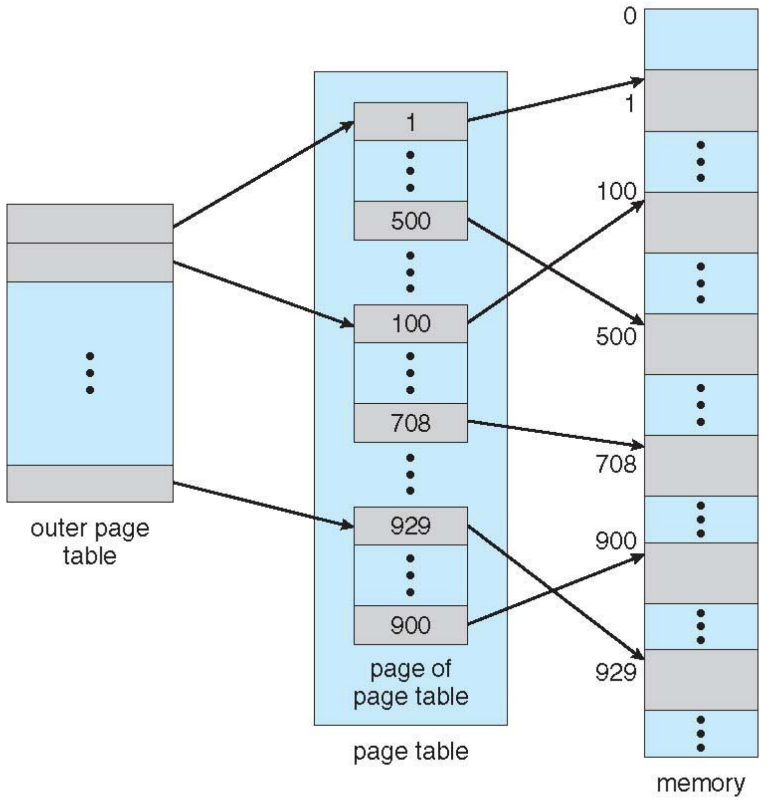


## Structure of the Page Table

- 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

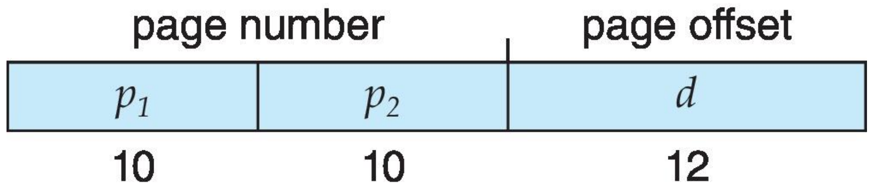
# Hierarchical Page Tables

- 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***



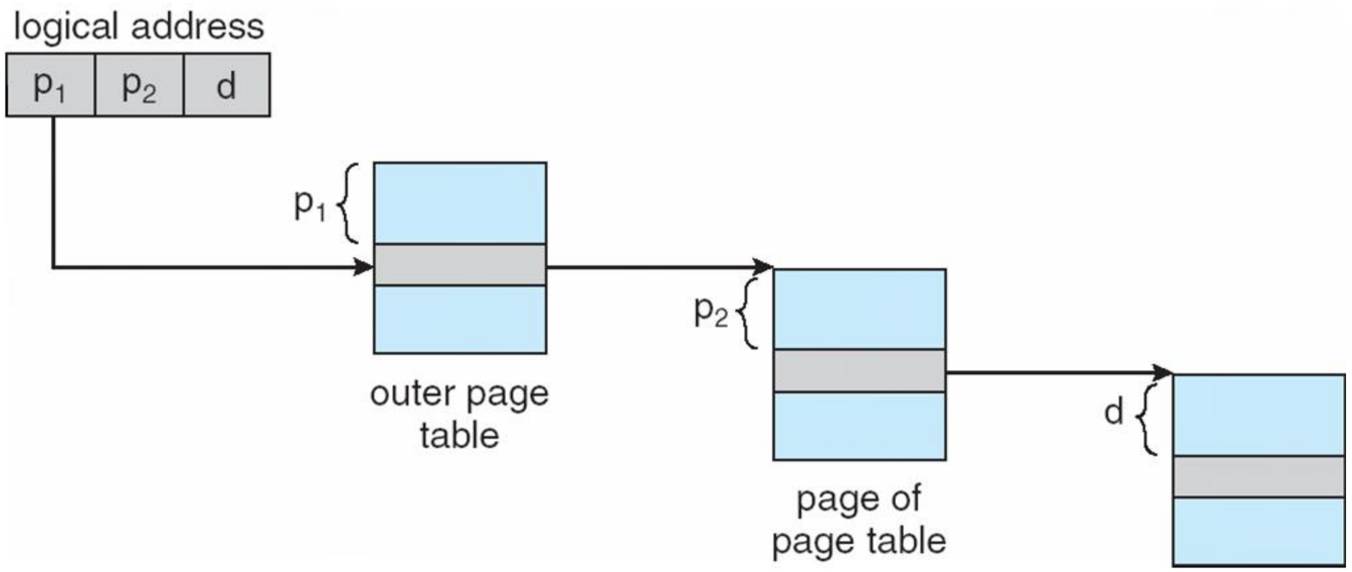
## Two-Level Paging Example

- A logical address (on 32-bit machine with 4KB 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:



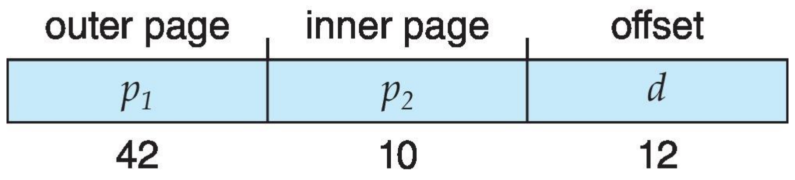
- where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

## Address-Translation Scheme



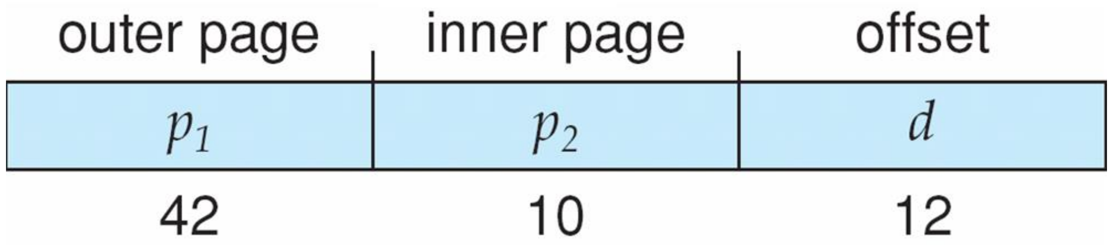
# 64-bit Logical Address Space

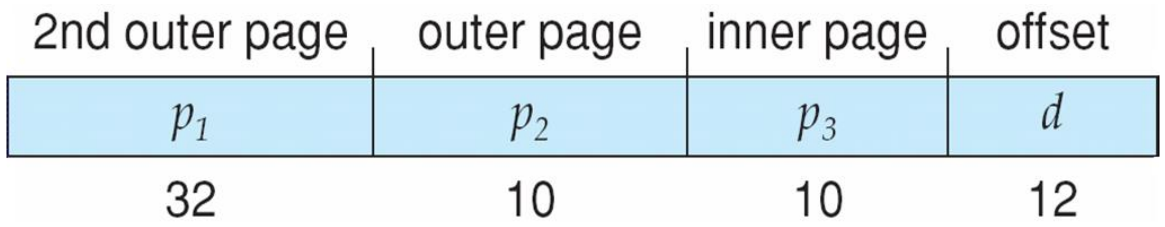
- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
  - Then page table has $2^{52}$ entries
  - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  - Address would look like



  - Outer page table has $2^{42}$ entries or $2^{44}$ bytes
  - One solution is to add a 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

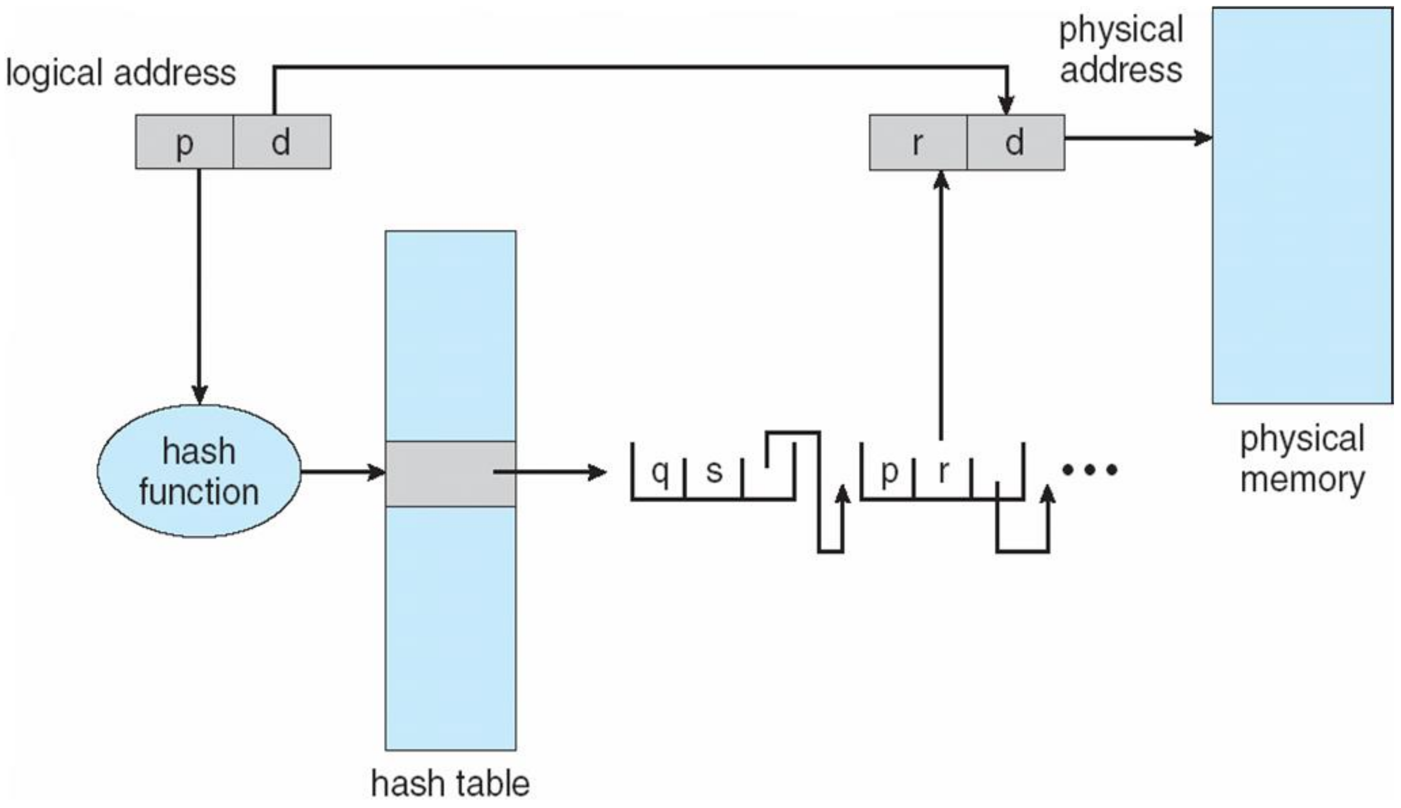




## Hashed Page Tables

- 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
  - If a match is found, the corresponding physical frame is extracted

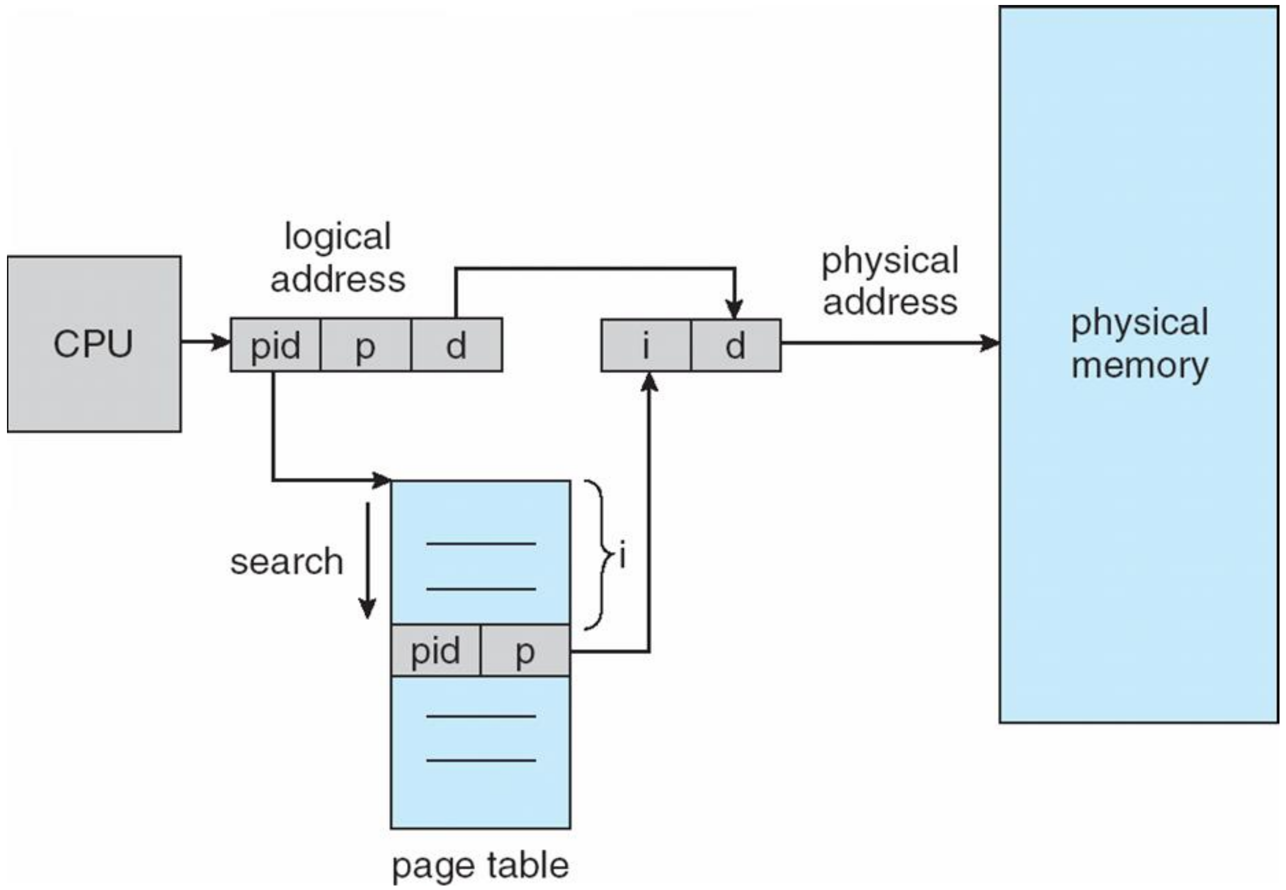
# Hashed Page Table



## 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



# Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on complex hashing
- 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)

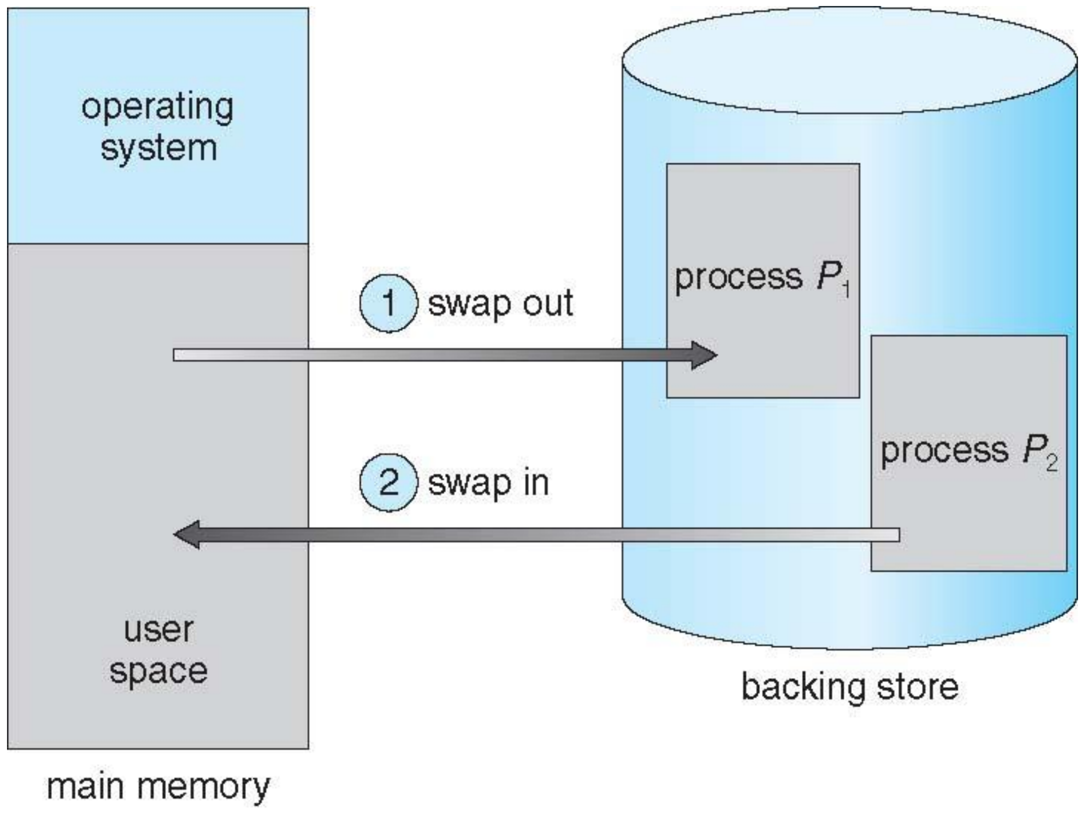
# Oracle SPARC Solaris (Cont.)

- 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 by loading the TTE into the TLB.

## Swapping

- A process (or a portion of 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

# Schematic View of Swapping



- Does the swapped out process need to swap back in to same physical addresses?
- What if I/O is pending for the process to be swapped out?

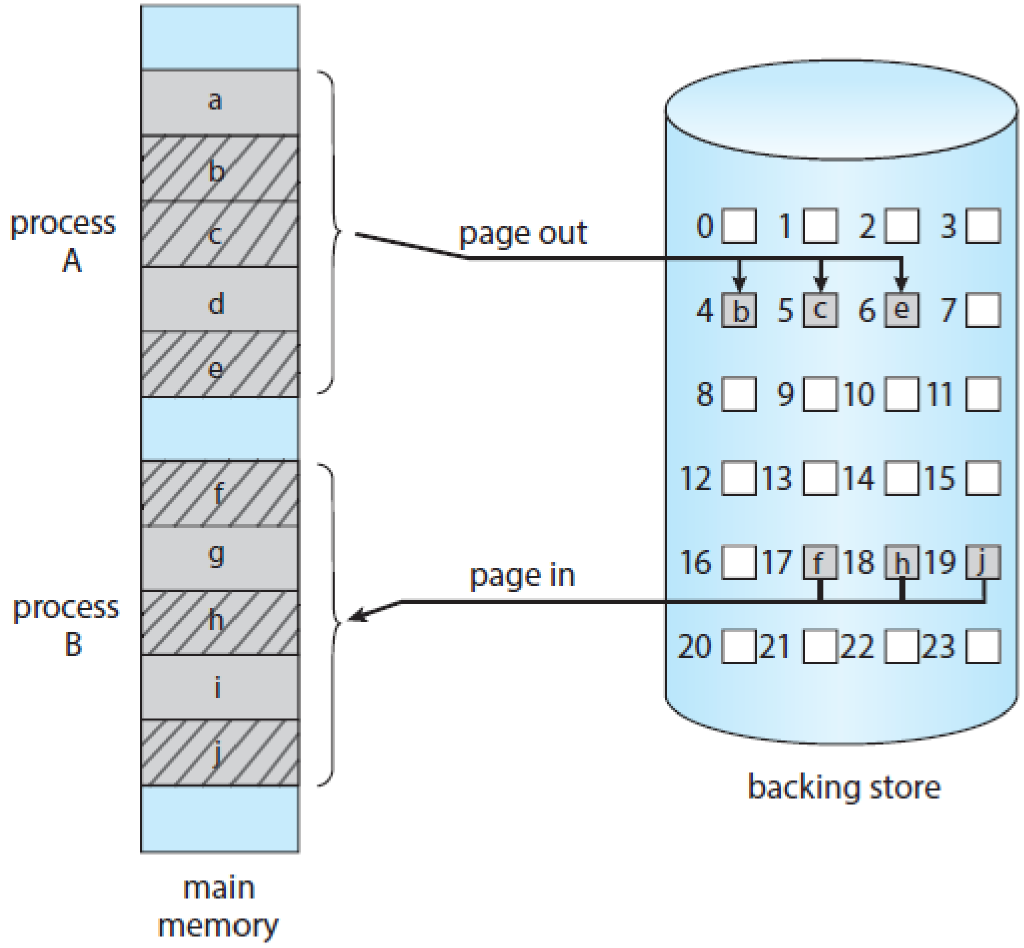
## Standard Swapping

- Moving entire processes between main memory and a backing store.
- When a process is swapped to the backing store
  - the data structures associated with the process must be written to the backing store.
  - For a multithreaded process
    - per-thread data structures must be swapped
- The OS must also maintain metadata for processes that have been swapped out.
- Advantage: allows physical memory to be oversubscribed
- Mostly idle processes are good candidates for swapping
- Disadvantage: Transfer Time

## Swapping with Paging

- Standard swapping was used in traditional UNIX systems
    - No longer used in contemporary OSs except Solaris
        - Solaris uses standard swapping in case of extremely low memory
- Most contemporary OS Including Linux and Windows uses swapping with pages.
    - small number of pages will be involved in swapping.
- **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**.

# Schematic View of Swapping with Paging

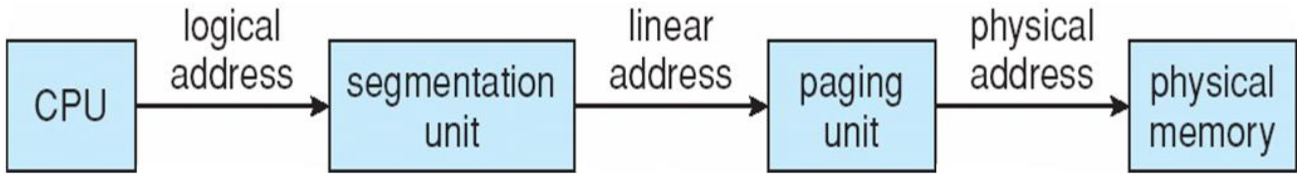


# Swapping on Mobile Systems

- 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

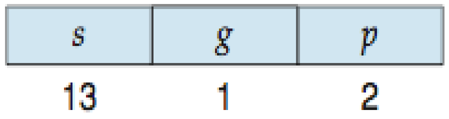
## Example: The Intel IA-32 Architecture

- 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.)

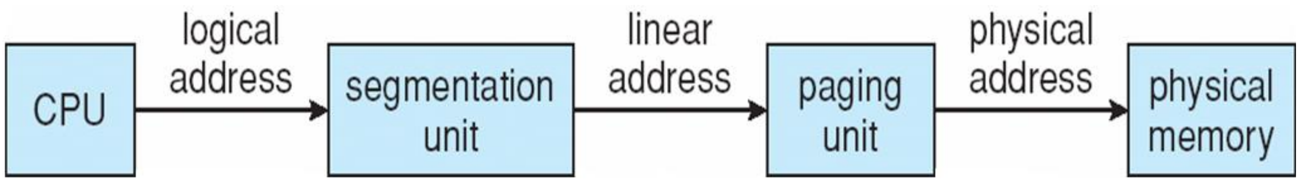
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses

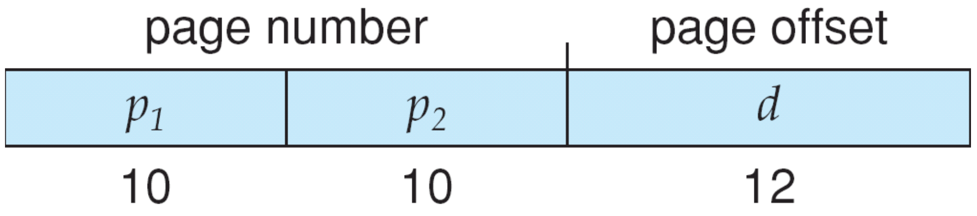


  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

process of address translation in a segmented and paged memory management system, particularly in the context of the Intel IA-32 architecture.

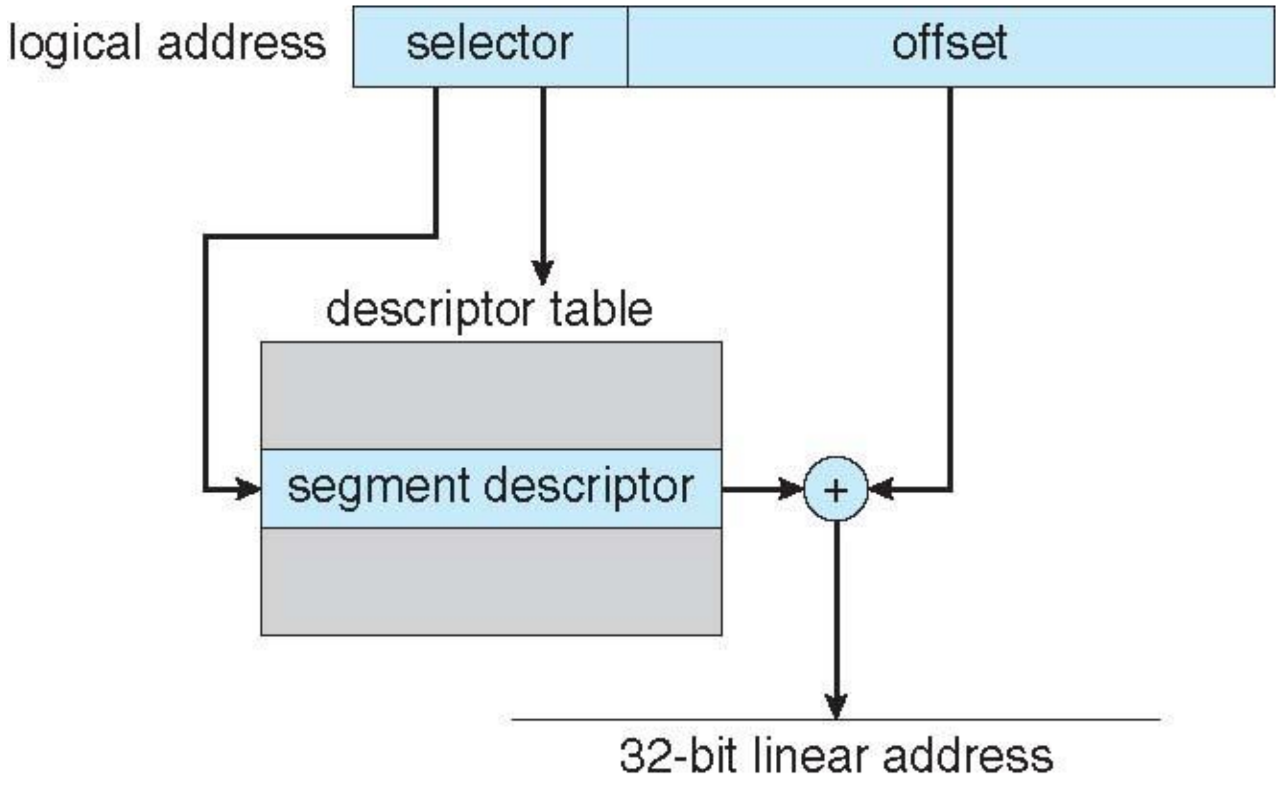
## Logical to Physical Address Translation in IA-32





Segmentation is a memory management technique used in computer systems that divides a program's memory into different segments based on logical divisions.

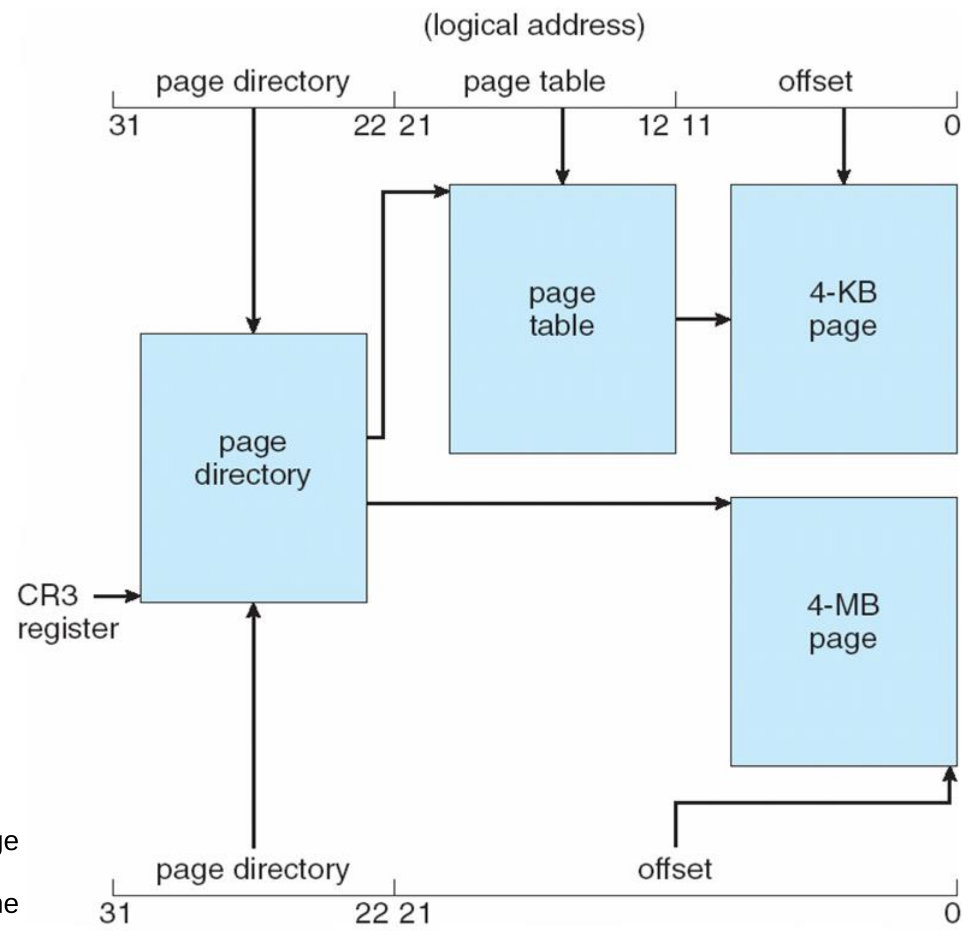
# Intel IA-32 Segmentation



# Intel IA-32 Paging Architecture

Page Directory and Page Tables:

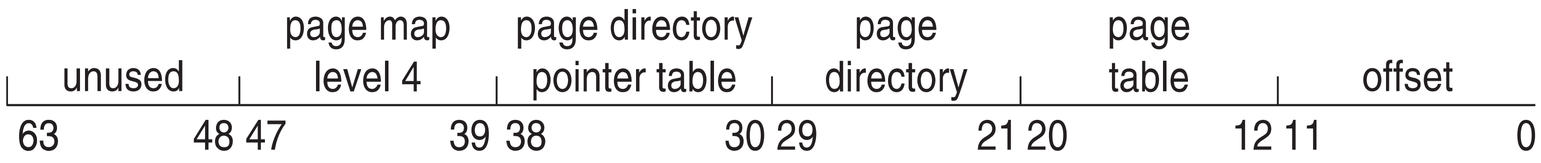
> IA-32 paging uses a two-level paging hierarchy:
Page Directory: The first level contains 1,024 entries (4 bytes each), pointing to Page Tables.
Page Tables: The second level also has 1,024 entries, with each entry pointing to a page frame in physical memory.
Each page in IA-32 is 4 KB, which provides fine-grained control over memory.



Virtual Address Breakdown: A 32-bit virtual address is divided into three parts:
10-bit Page Directory Index: Points to an entry in the Page Directory.
10-bit Page Table Index: Points to an entry in the Page Table.
12-bit Offset: Specifies the exact byte within the 4 KB page.
This breakdown allows the system to map 4 GB of virtual address space, broken into 4 KB pages.

# 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



## Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Self-Study.

# End of Chapter 9