
LUT
University
LAND
OF THE
CURIOUS

JANUARY 17, 2023

OPERATING SYSTEMS AND SYSTEMS PROGRAMMING (CT30A3370) 6 CREDITS

Venkata Marella

CHAPTER 8: MEMORY MANAGEMENT

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

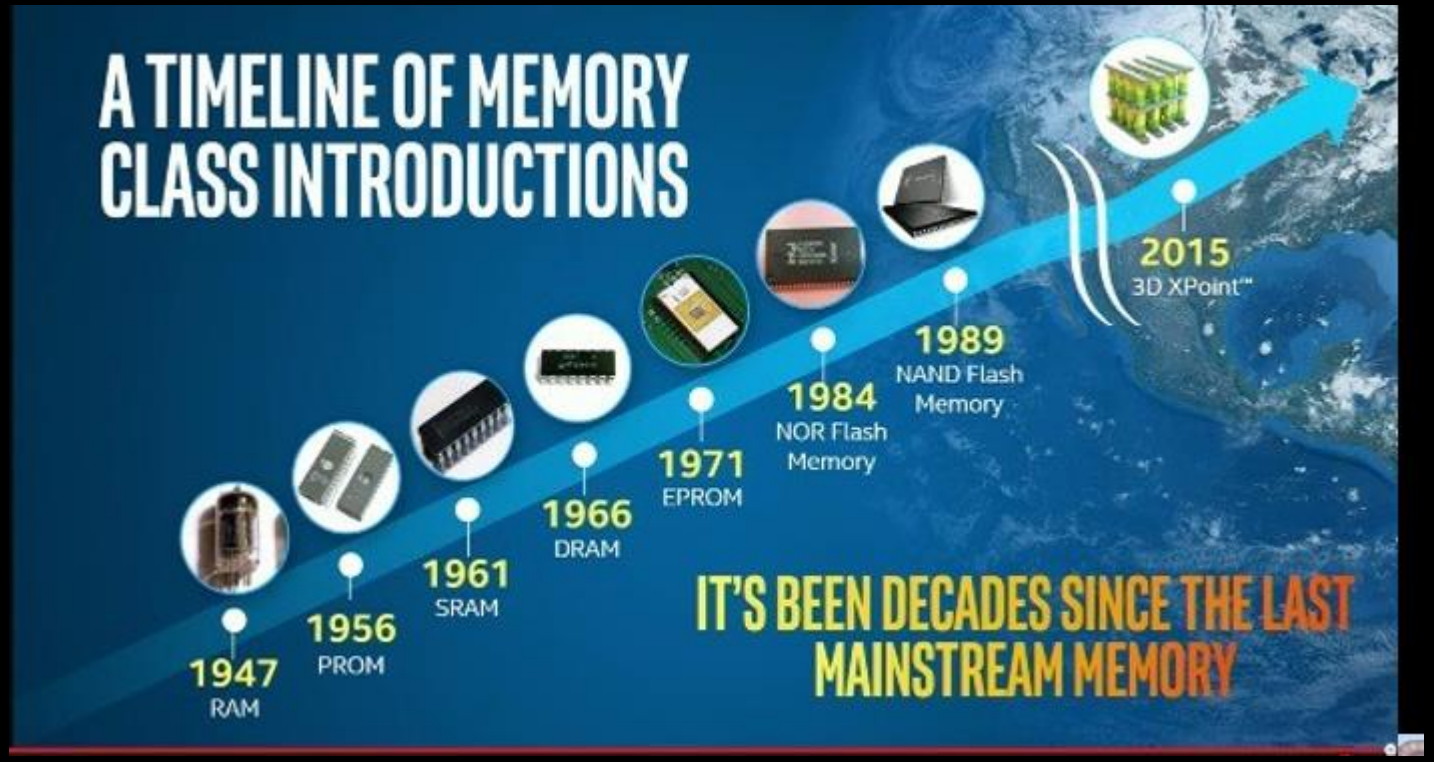
OBJECTIVES

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- 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
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

MEMORY HISTORY

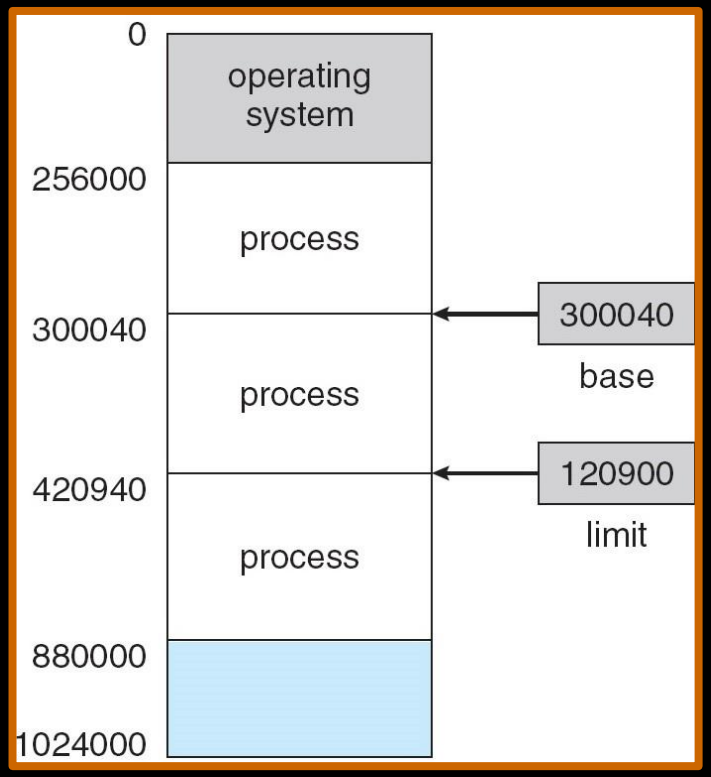


MEMORY HIERARCHY

	L1 Cache	L2 Cache	Main memory
Block size	16--32 bytes	32--64 bytes	4--16 KB
Size	16--64 KB	256KB—8MB	
Hit time	1 Clock Cycle	1—4 Clock Cycles	10—40 Clock Cycles
Backing Store	L2 Cache	Main memory	Disk
Block replacement	Random	Random	Replacement strategies
Miss penalty	4-20 clock cycles	40-200 clock cycles	~6M clock cycles

BASE AND LIMIT REGISTERS

- A pair of **base** and **limit** registers define the logical address space

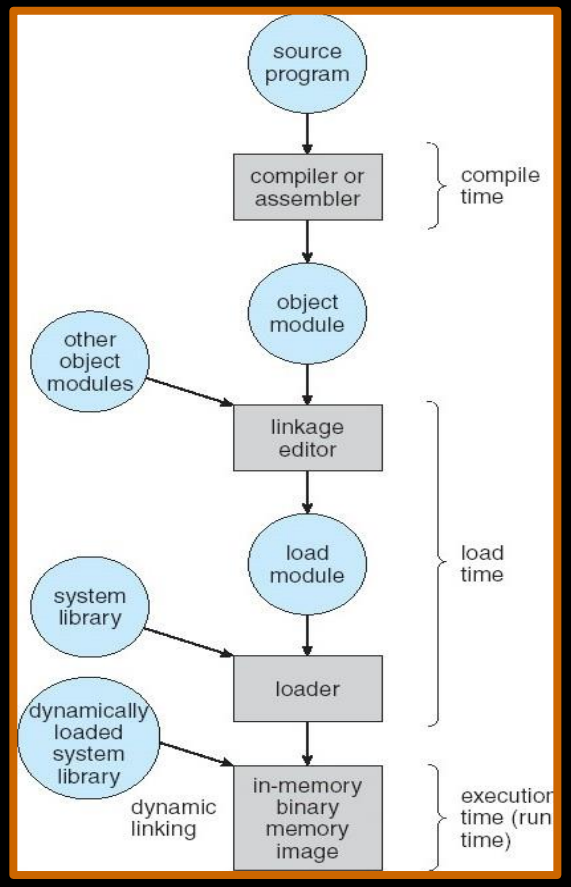


BINDING OF INSTRUCTIONS AND DATA TO MEMORY

Mapping From one address space to another

- 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



A **compiler** is a computer program (or set of programs) that transforms source code written in a computer language (the ***source language***) into another computer language (the ***target language***, often having a binary form known as ***object code***).

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program. On Unix variants the term **loader** is often used as a synonym for linker.

LOGICAL VS. PHYSICAL ADDRESS SPACE

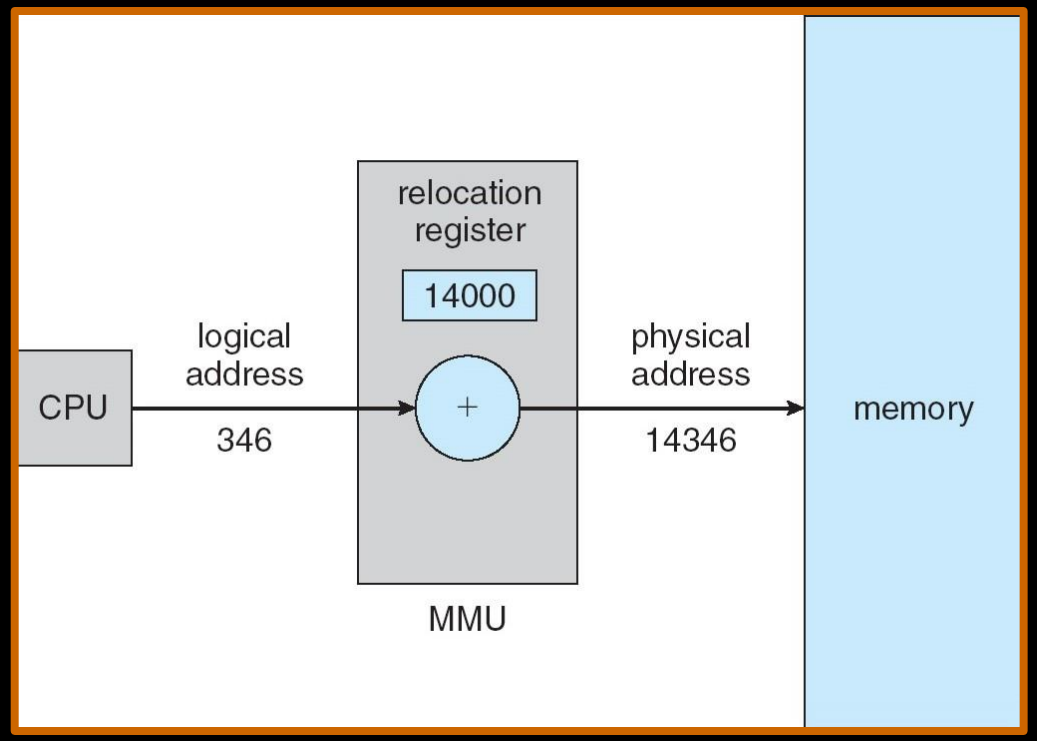
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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

MEMORY-MANAGEMENT UNIT (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, 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 RELOCATION USING A RELOCATION REGISTER

A simple MMU



DYNAMIC LOADING

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- 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

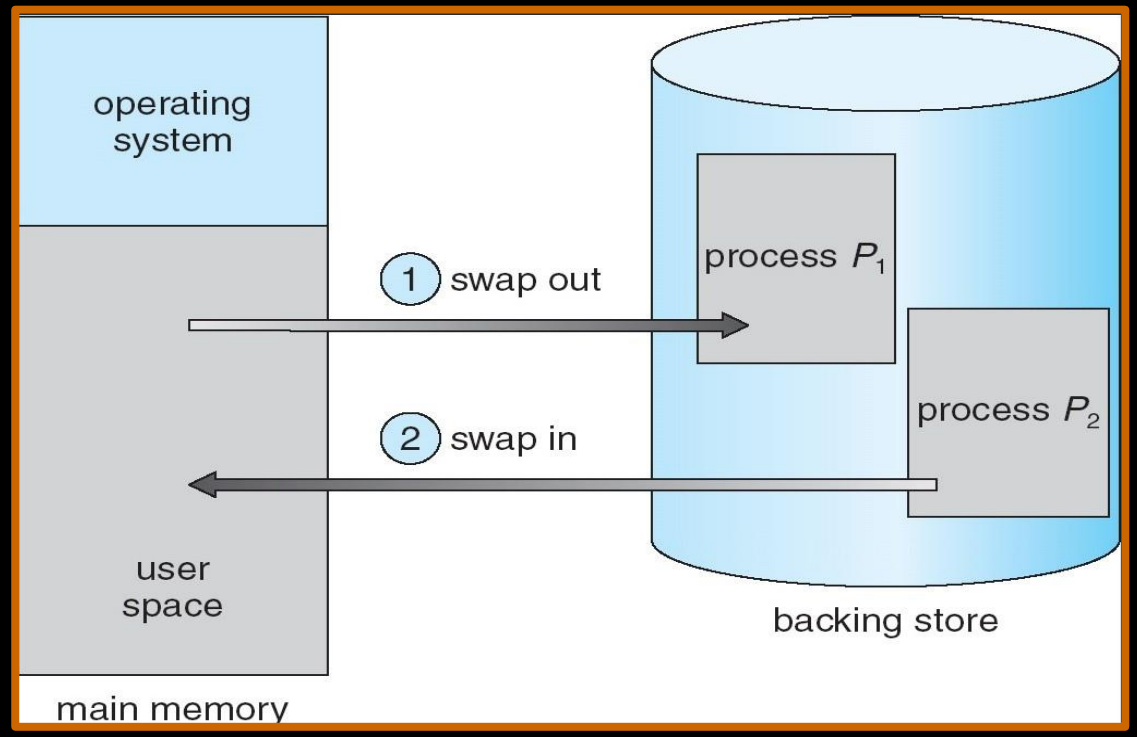
DYNAMIC LINKING

- 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 needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Relinking of new library not needed

SWAPPING

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **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
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

SCHEMATIC VIEW OF SWAPPING

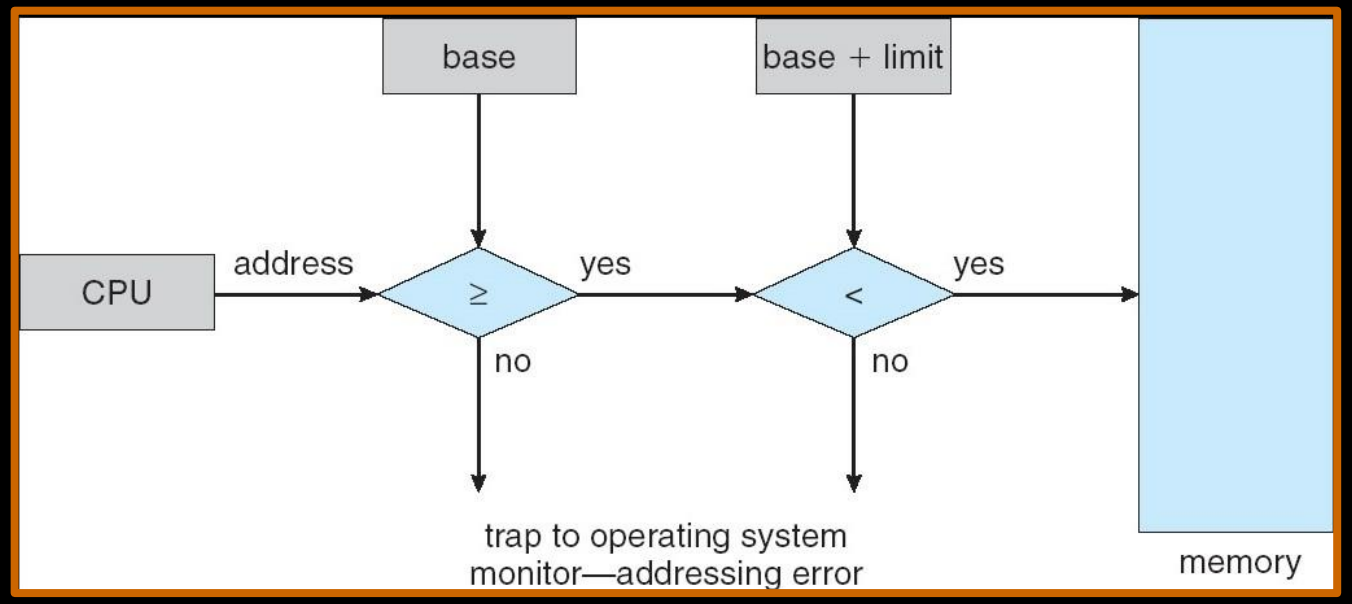


CONTIGUOUS ALLOCATION

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory

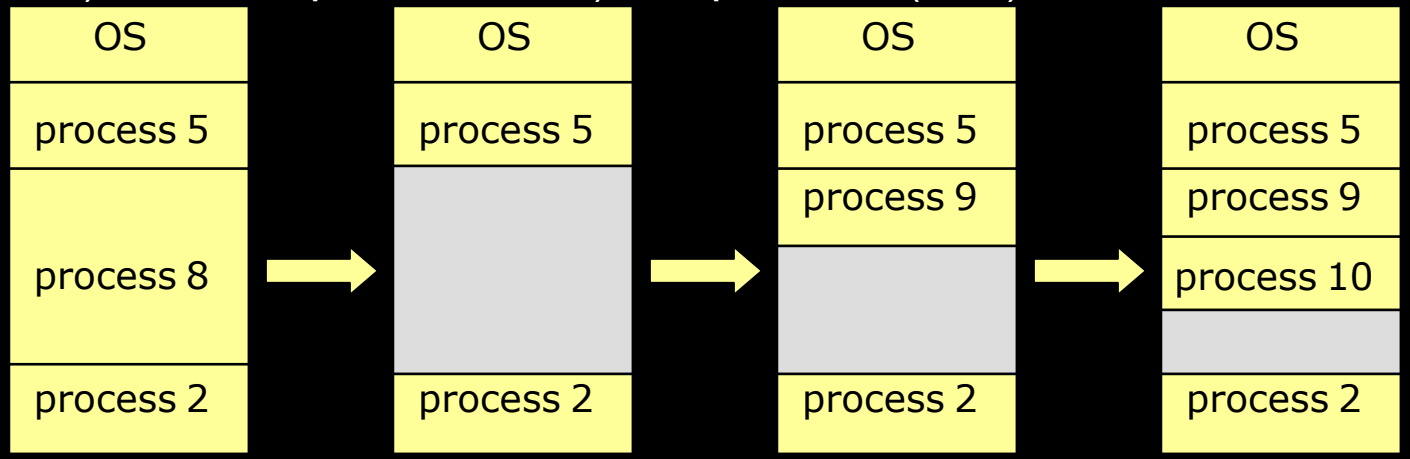
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest *physical* address
  - Limit register contains range of *logical* addresses – each logical address must be less than the limit register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

HW ADDRESS PROTECTION WITH BASE AND LIMIT REGISTERS



CONTIGUOUS ALLOCATION (CONT.)

- Multiple-partition allocation
  - 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
    - 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

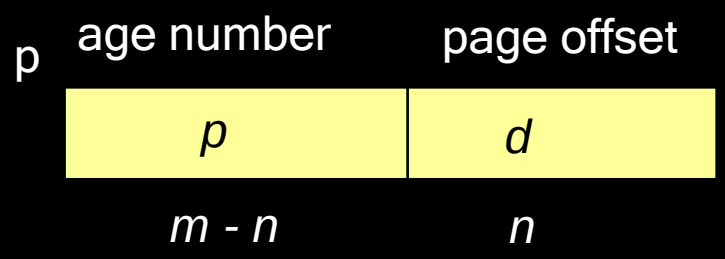
INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
When a process is allocated more memory than required few space is left in the block which causes INTERNAL FRAGMENTATION.	After execution of processes when they are swapped out of memory and smaller processes replace them many small non contiguous blocks of empty spaces are formed which can serve a new request if put together but as they are non contiguous they cannot service a new request causing EXTERNAL FRAGMENTATION.
It occurs when memory is divided into fixed sized partitions.	It occurs when memory is divided into variable sized partitions based on the size of process.
It can be cured by allocating memory dynamically or having partitions of different sizes..	It can be cured by compaction,paging and segmentation.

PAGING

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames**
  (size is power of 2, between 512 bytes and 8,192 bytes)
- 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

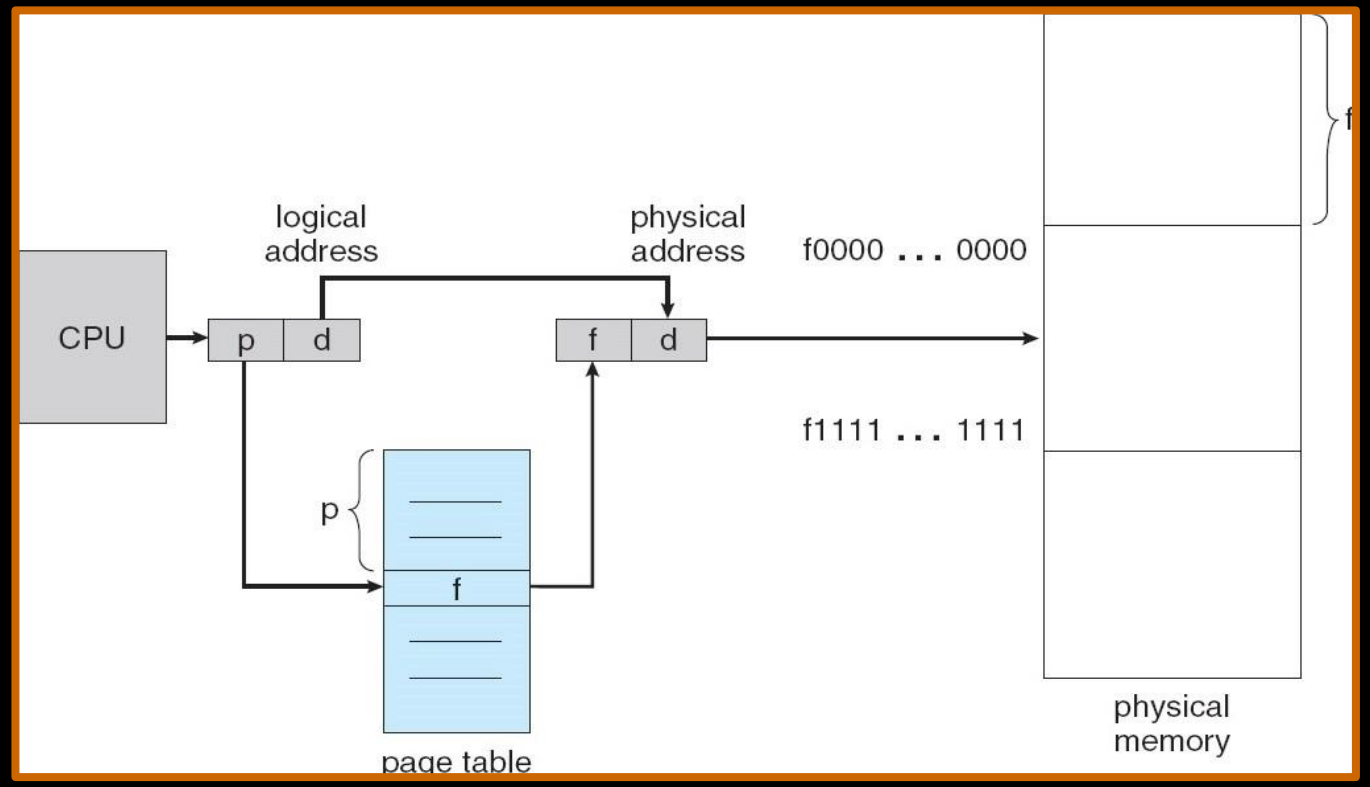
ADDRESS TRANSLATION SCHEME

- 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

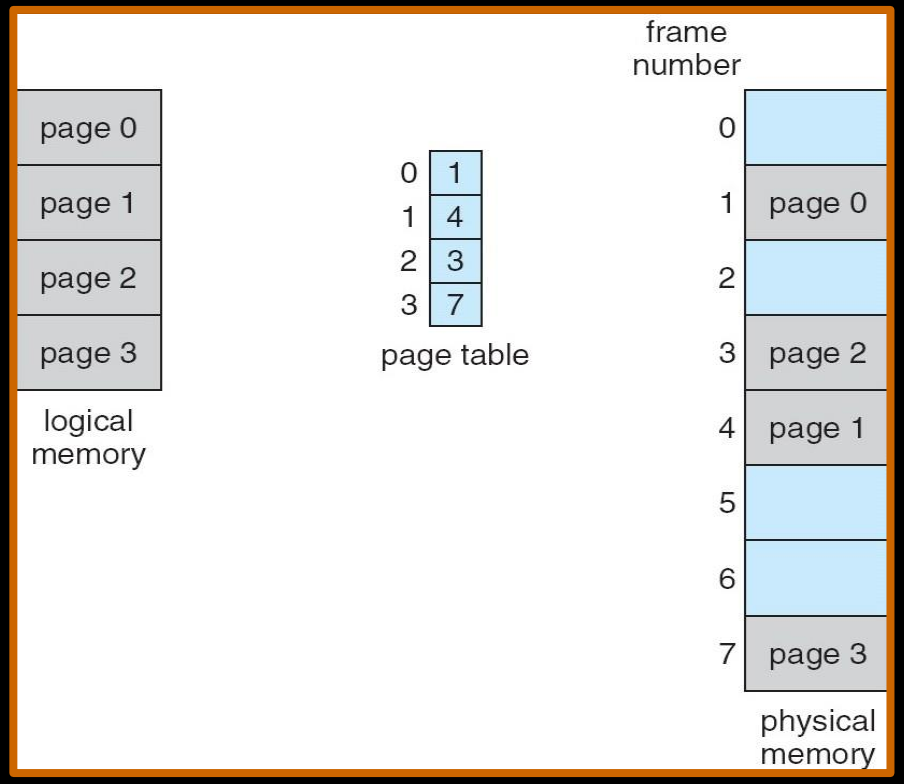


  - For given logical address space 2^m *and page size* 2^n

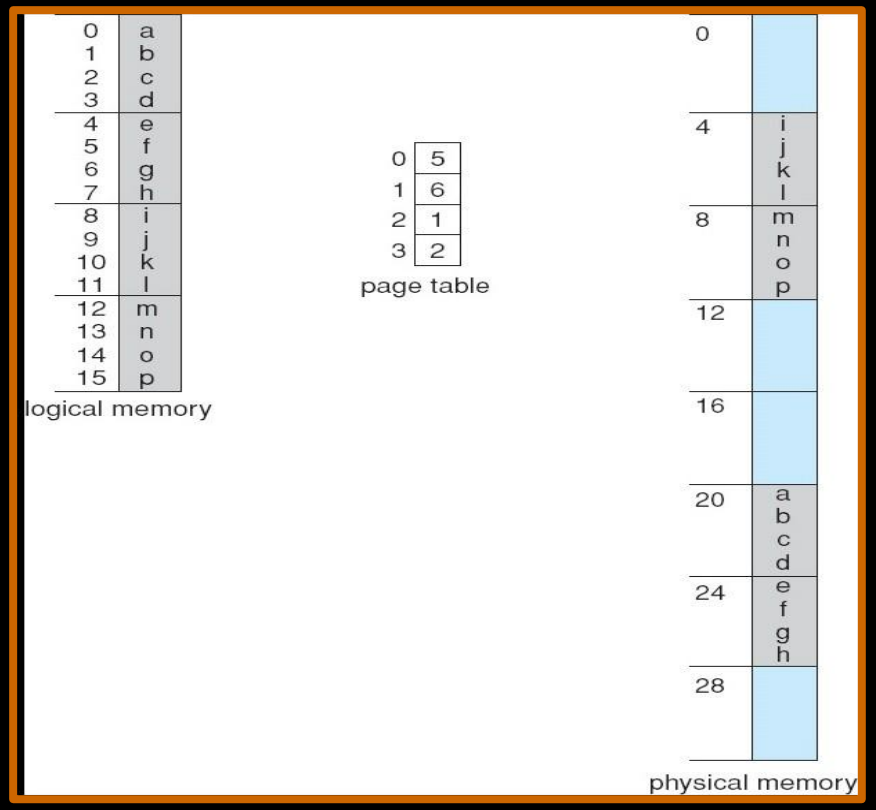
PAGING HARDWARE



PAGING MODEL OF LOGICAL AND PHYSICAL MEMORY

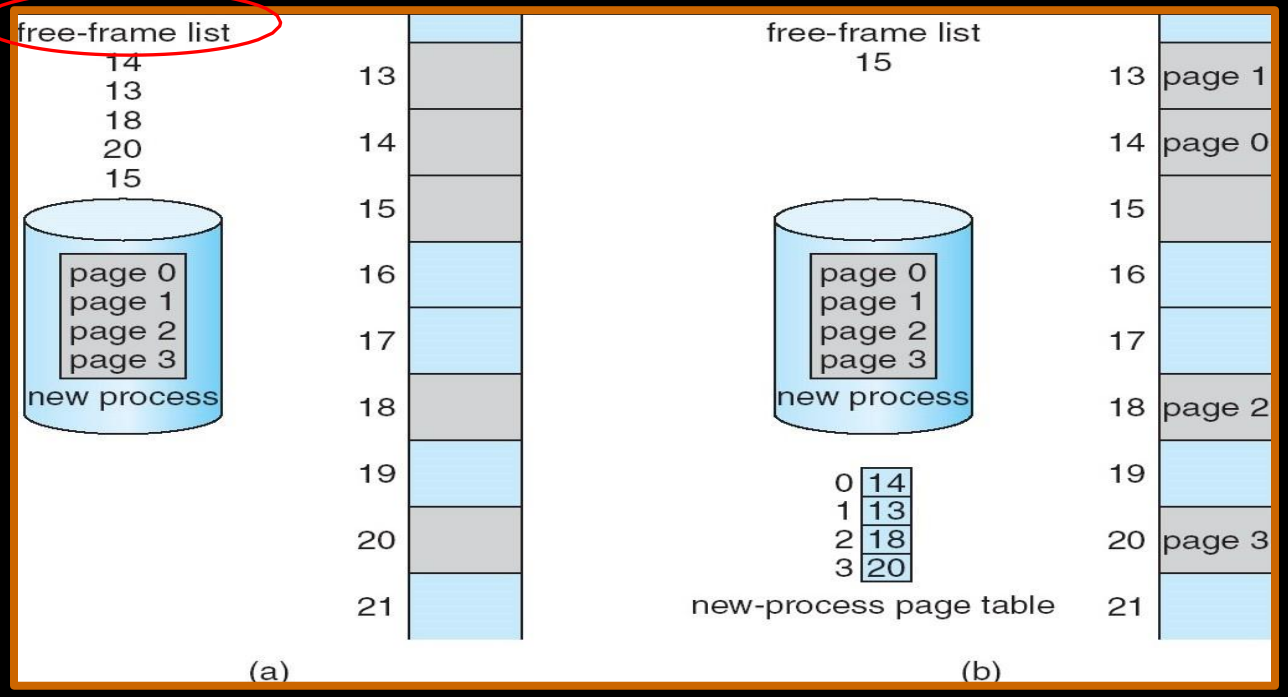


PAGING EXAMPLE



32-byte memory and 4-byte pages

FREE FRAMES



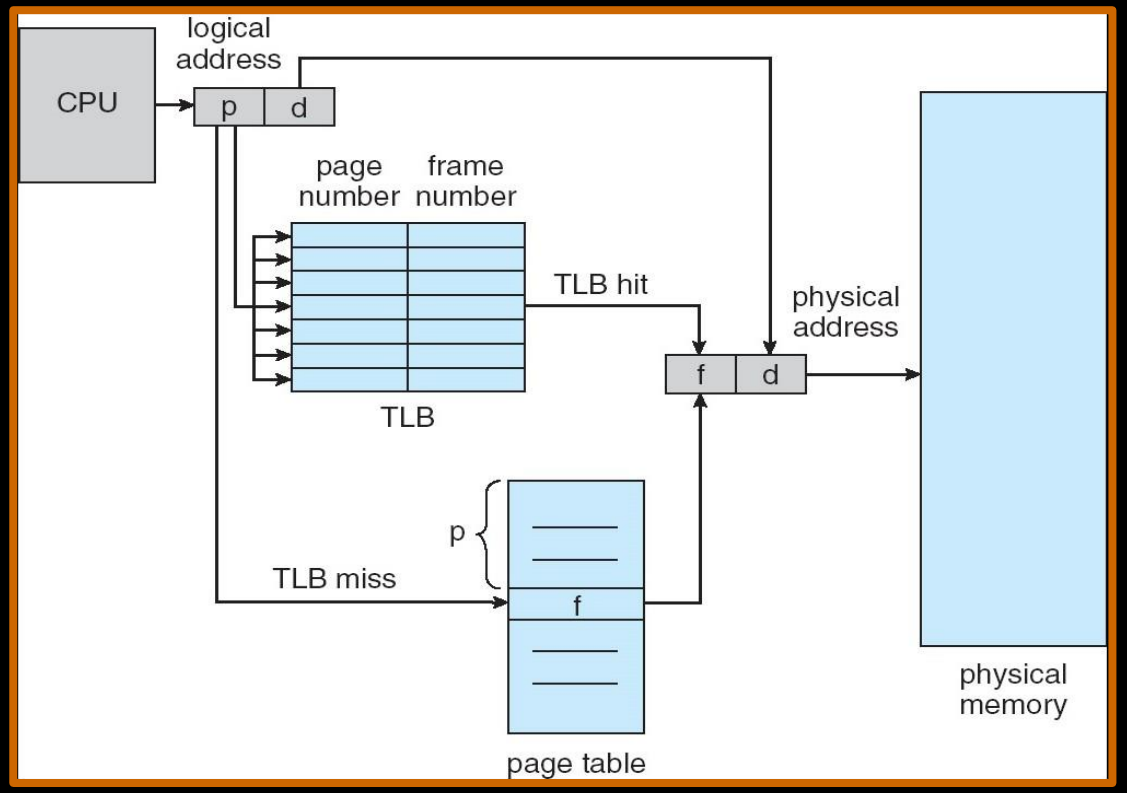
Before allocation

After allocation

HARDWARE 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 **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address- space protection for that process

PAGING HARDWARE WITH TLB



EFFECTIVE ACCESS TIME

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio - percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

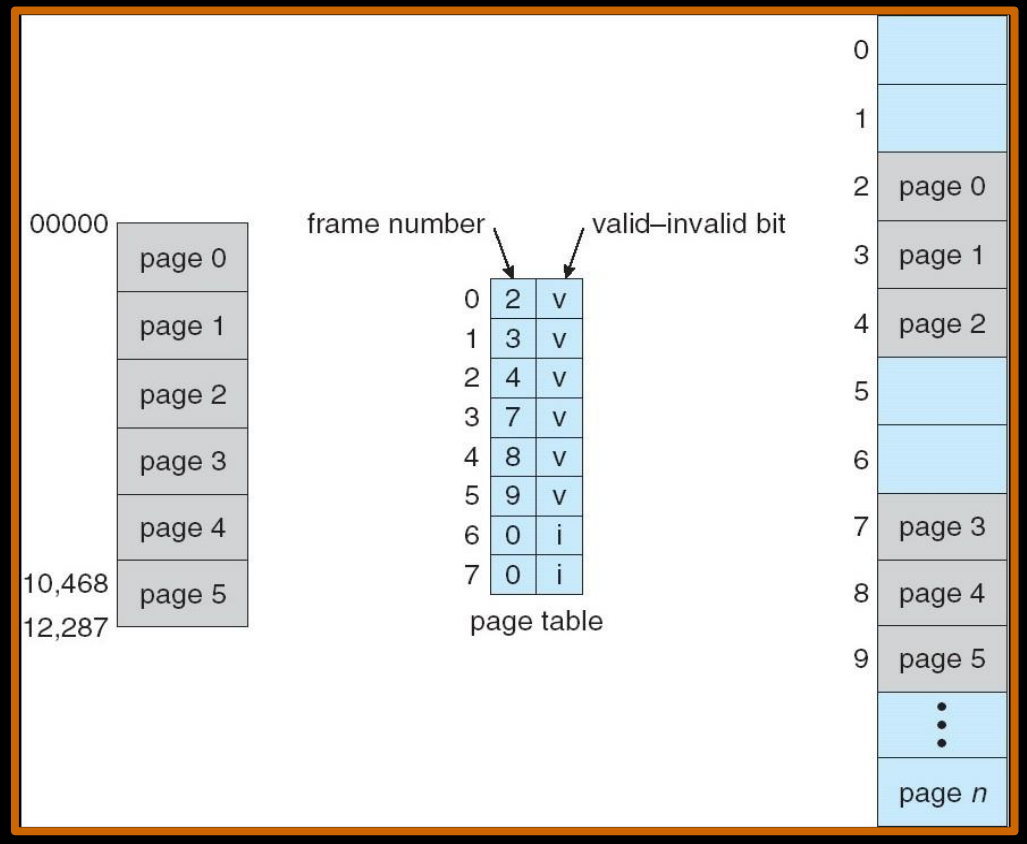
$$= 2 + \varepsilon - \alpha$$

MEMORY PROTECTION IN PAGED SCHEME

- Memory protection implemented by associating protection bit with each frame

- **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

VALID (V) OR INVALID (I) BIT IN A PAGE TABLE

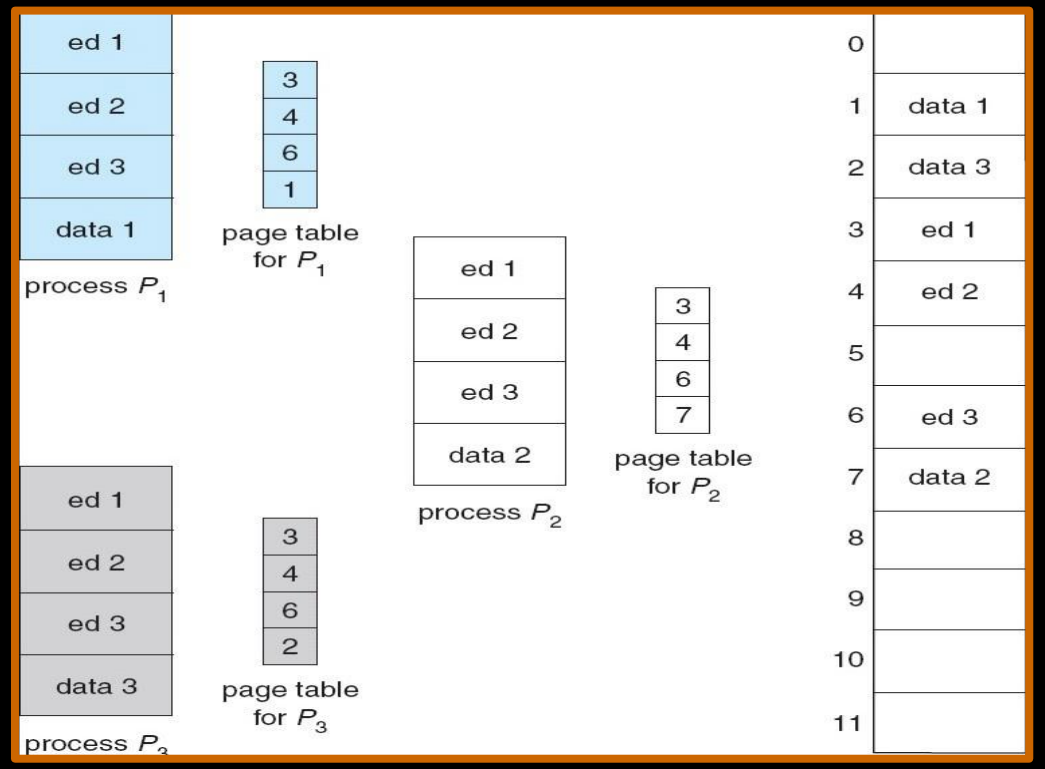


SHARED PAGES

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **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

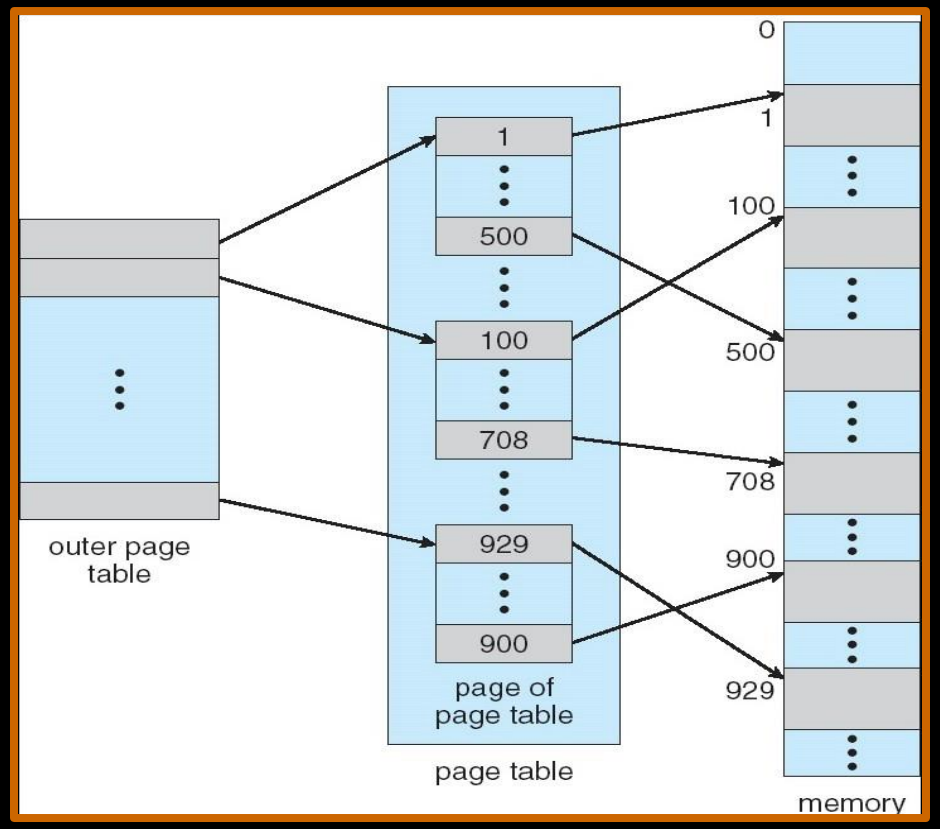
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

HIERARCHICAL PAGE TABLES

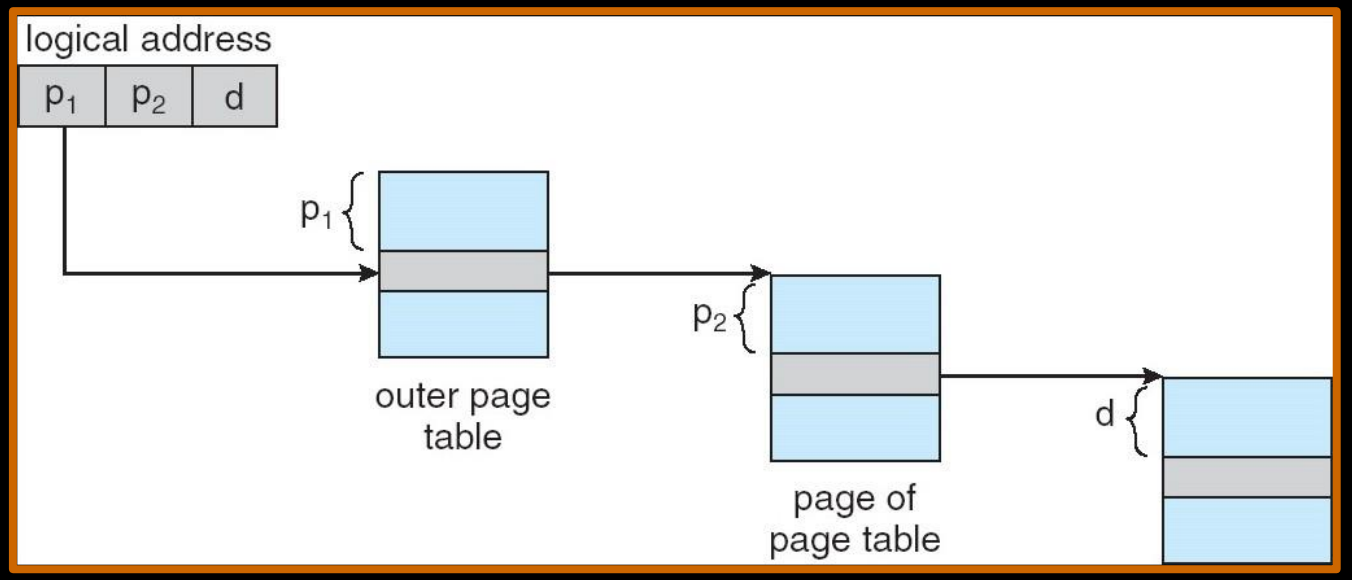
- Break up the logical address space into multiple page tables – to page the page table

- A simple technique is a two-level page table

TWO-LEVEL PAGE-TABLE SCHEME



ADDRESS-TRANSLATION SCHEME



THREE-LEVEL PAGING SCHEME

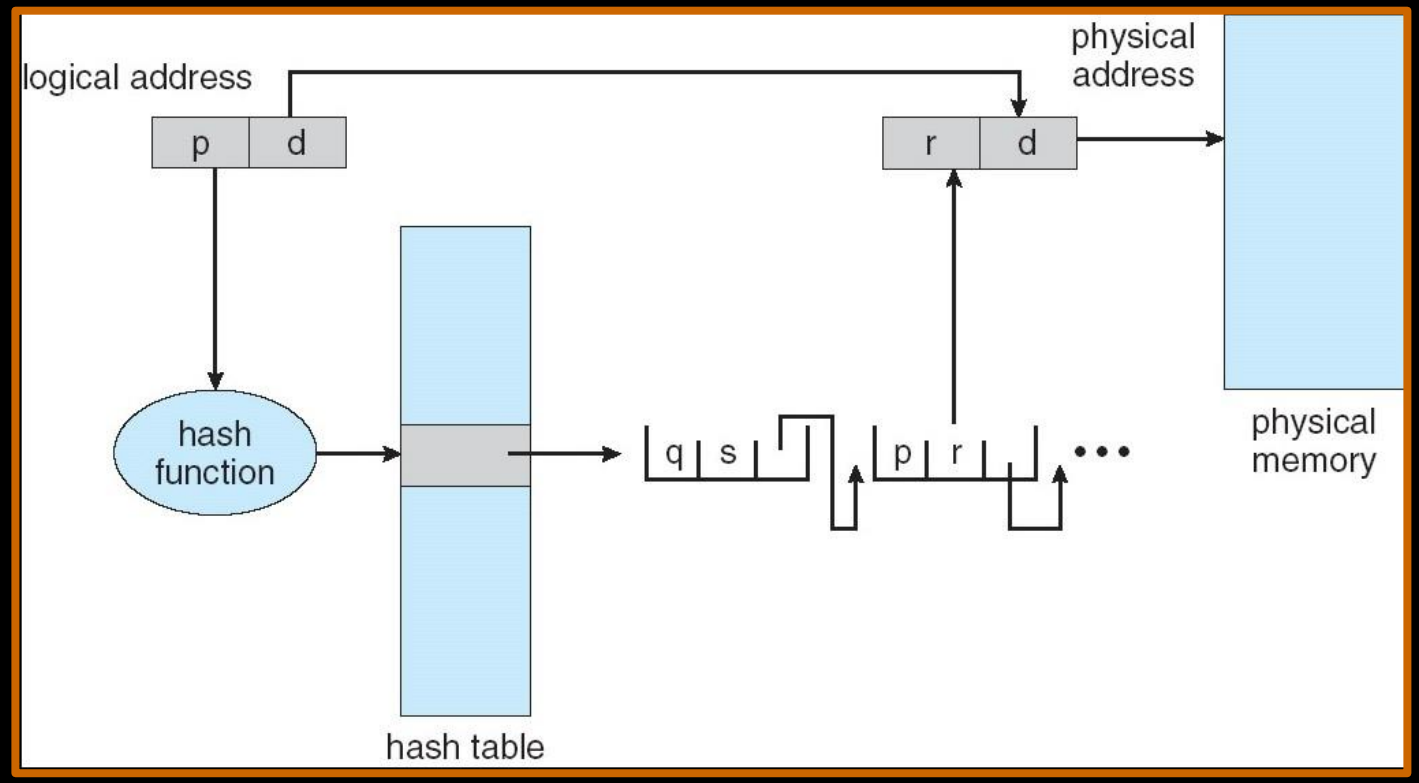
outer page	inner page	offset
p_1	p_2	d
42	10	12

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.
- 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

Only one page table in the whole system

- 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

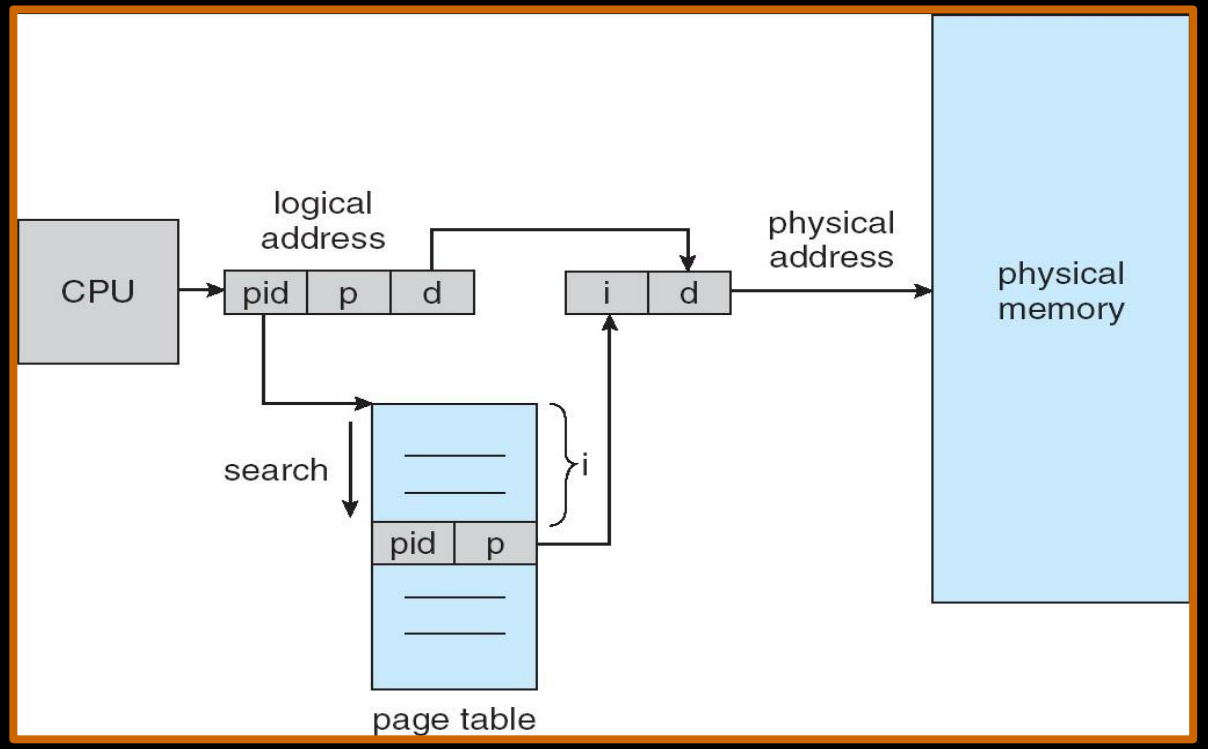
WHY INVERTED PAGE TABLE (IPT)

- Problem:
  - Page table overhead increases with address space size
  - Page tables get too big to fit in memory!
- Consider a computer with 64 bit addresses
  - Assume 4 Kbyte pages (12 bits for the offset)
  - Virtual address space = 2^{52} pages!
  - Page table needs 2^{52} entries!
  - This page table is too large for memory!
    - Many peta-bytes per process page table

WHY INVERTED PAGE TABLE (IPT)

- How many mappings do we need (maximum) at any time?
- We only need mappings for pages that are in memory!
  - A 256 Kbyte memory can only hold 64 4Kbyte pages Only need 64 page table entries on this computer!
- An inverted page table
  - Has one entry for every frame of memory
  - Records which page is in that frame
  - Is indexed by frame number not page number!
  - So how can we search an inverted page table on a TLB miss fault?

INVERTED PAGE TABLE ARCHITECTURE



Search is slow, so put page table entries into a hash table.
TLB can be used to speed up hash-table reference.

SEGMENTATION

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

  main program,

  procedure,

  function,

  method, object,

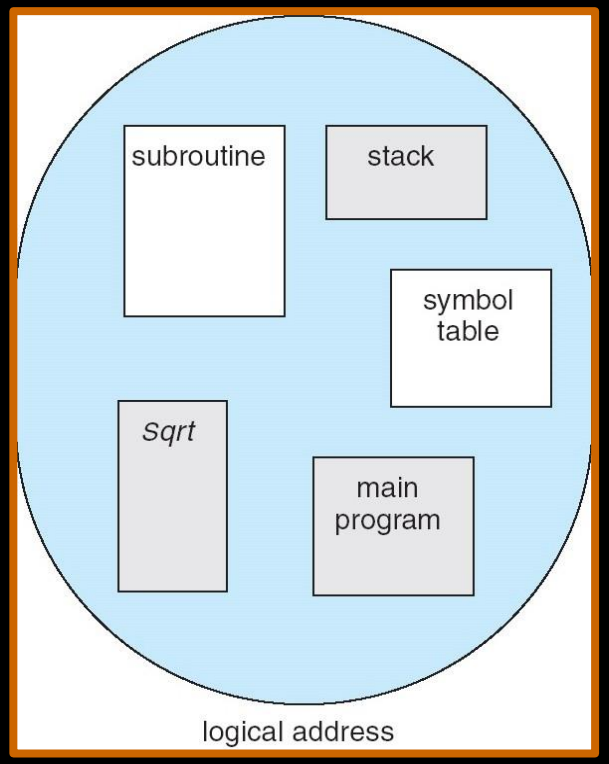
  local variables, global variables,

  common block,

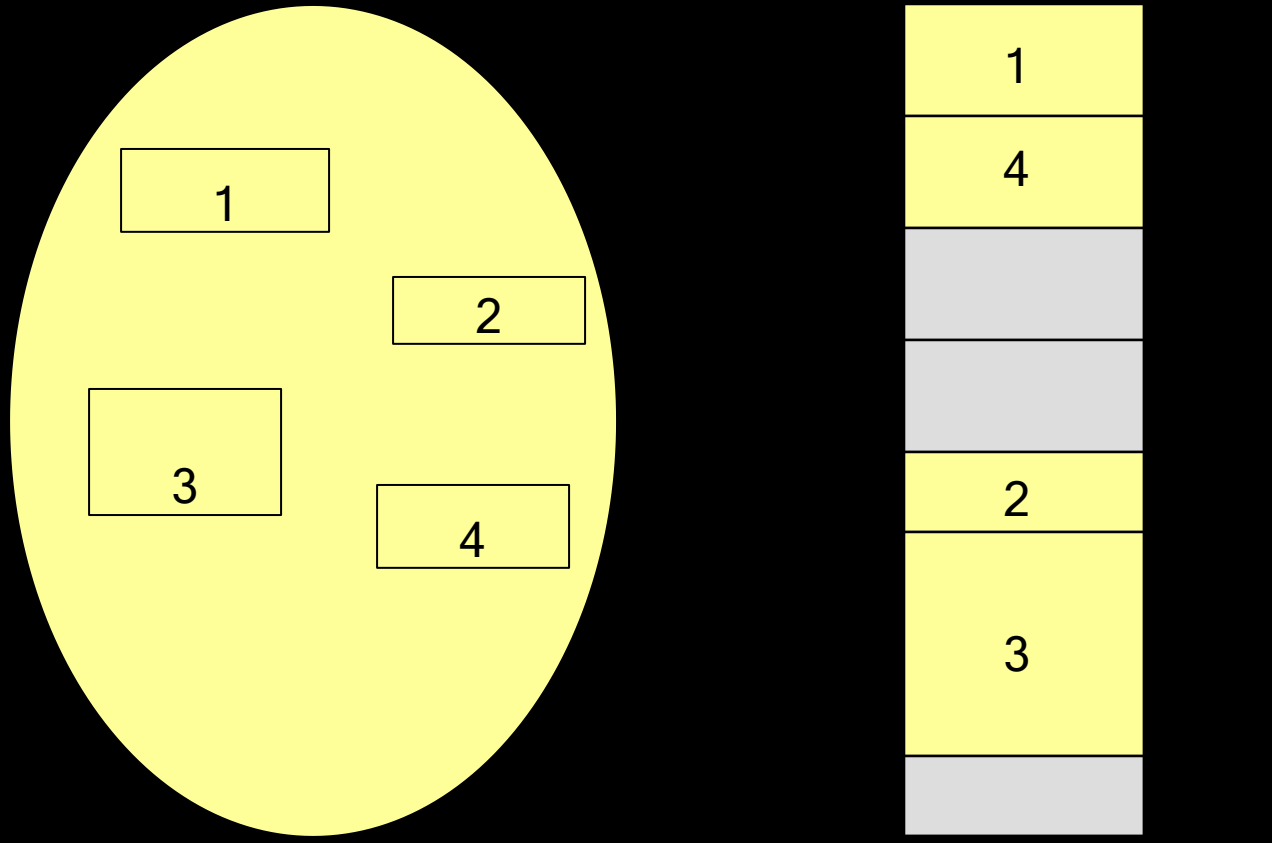
  stack,

  symbol table, arrays

USER'S VIEW OF A PROGRAM



LOGICAL VIEW OF SEGMENTATION



SEGMENTATION ARCHITECTURE

- Logical address consists of a two tuple:

  <segment-number, offset>,

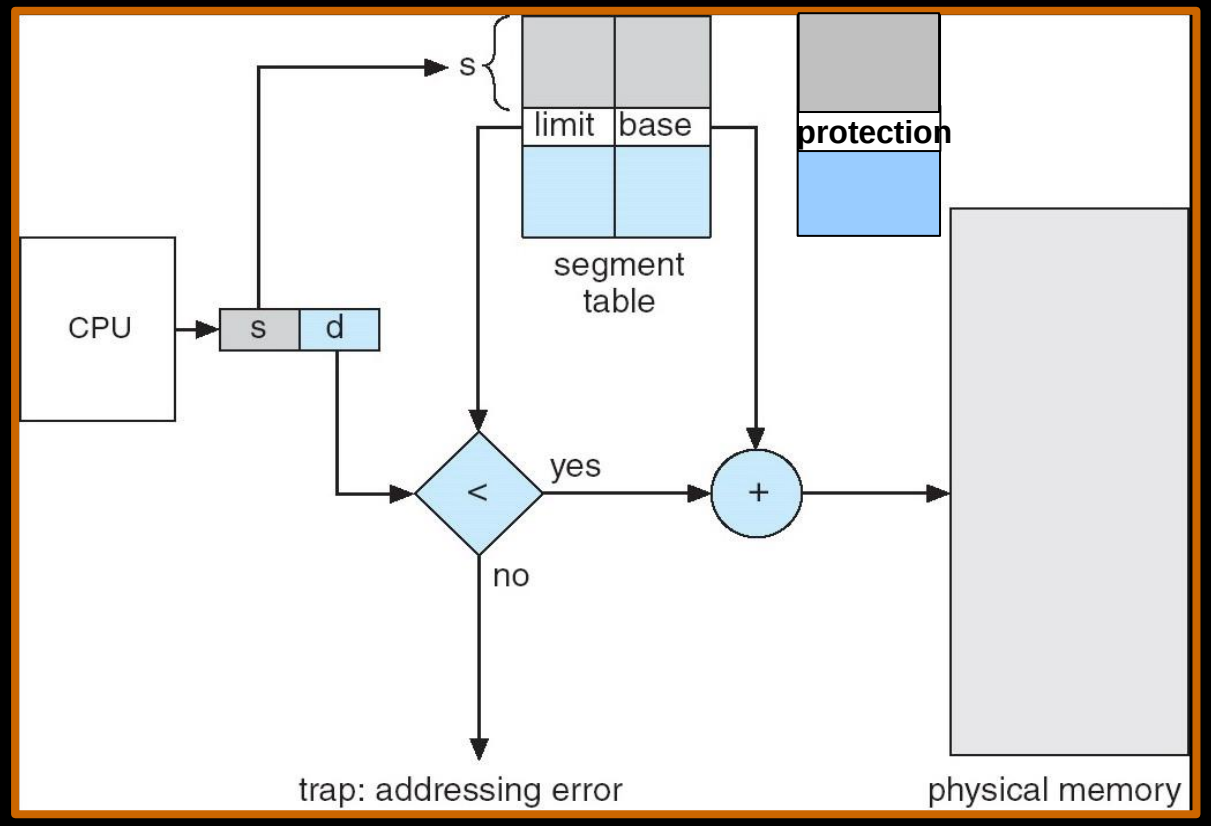
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number **s** is legal if **s** < **STLR**

SEGMENTATION ARCHITECTURE (CONT.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

SEGMENTATION HARDWARE



SEGMENTATION VS. PAGING

Segment is good logical unit of information
- sharing, protection

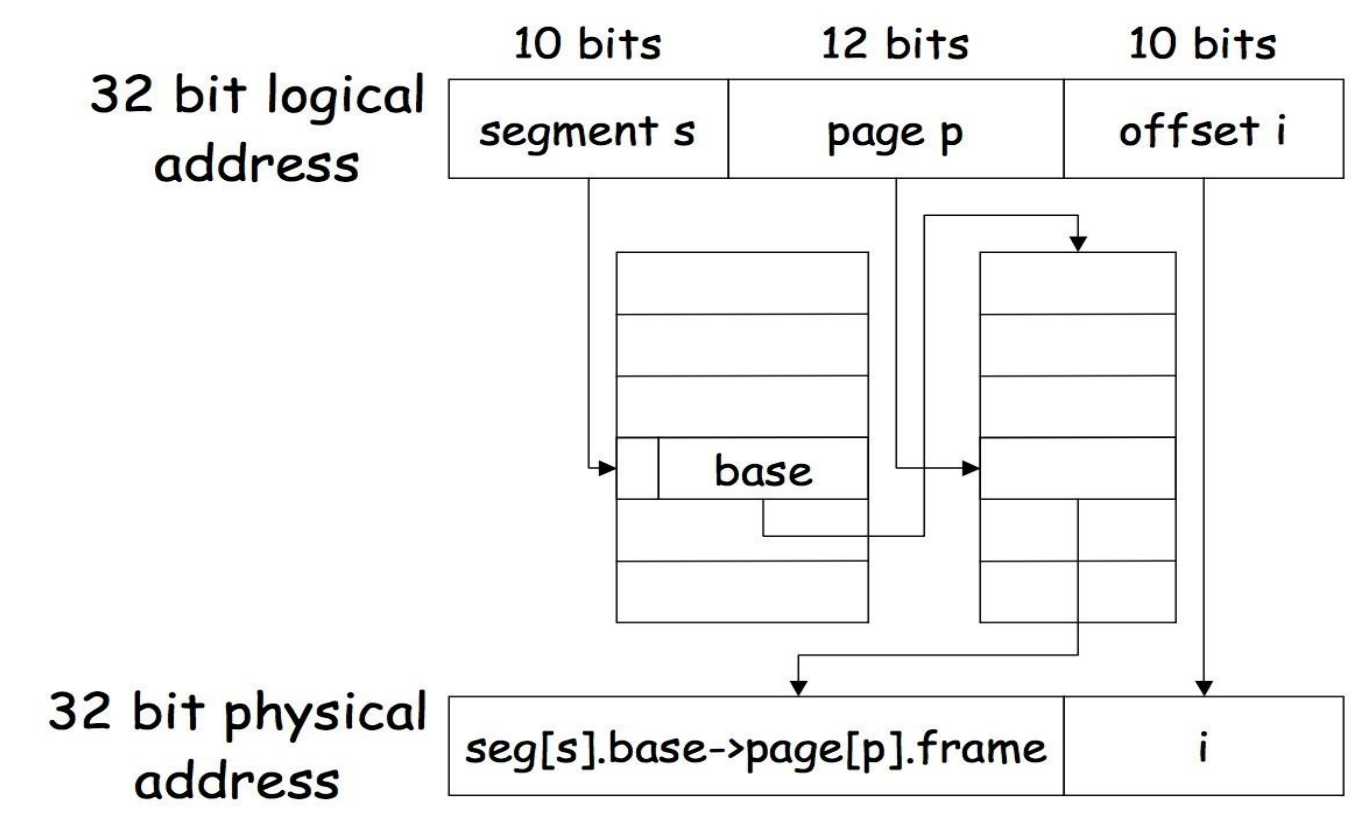
Page is good physical unit of information
- simple memory management

Best of both
- segmentation on top of paging

SEGMENTATION ON PAGING

- Logical memory is composed of segments
- Each segment is composed of pages
- Segment table
  - per process, in memory pointed to by register
  - entries map seg # to page table base
  - shared segment: entry points to shared page table
- Page tables (like before)

SEGMENTATION + PAGING



END OF CHAPTER 8