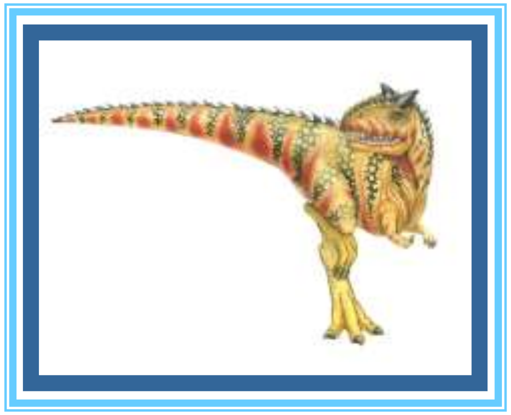
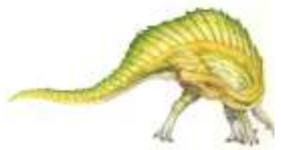
Chapter 8: Main Memory



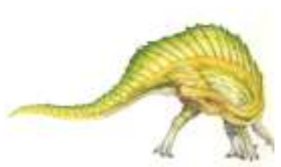
Chapter 8: Memory Management

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



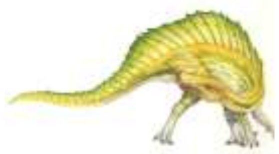
Objectives

- 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 the only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing CPU **stall(**A **stall** happens when the CPU **has nothing to execute** because it's waiting for memory.**)**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
  - correct operation to protect the operating system from access by user processes
  - protect user processes from one another



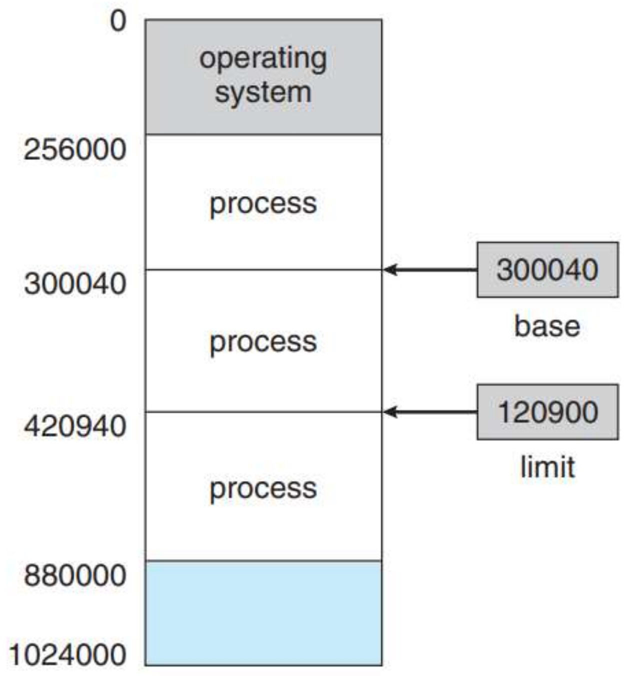
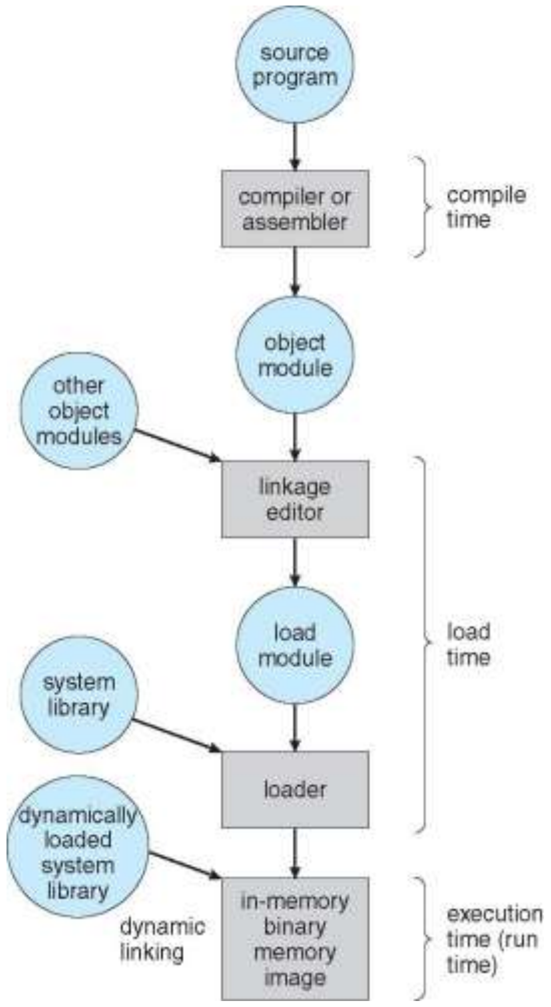


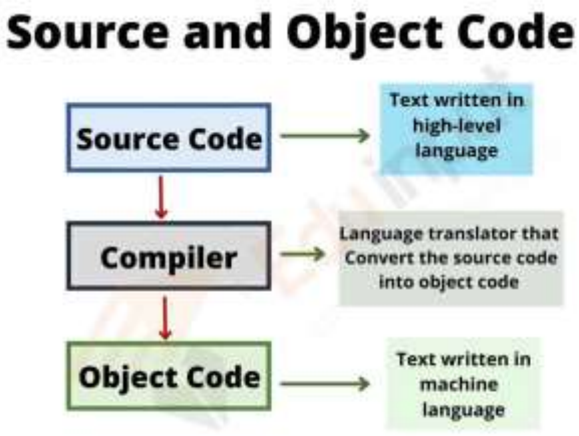
Figure 8.1 A base and a limit register define a logical address space.

The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error

Multistep Processing of a User Program

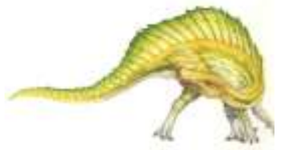



Source and Object Code
Source Code
Text written in high-level language
Compiler
Language translator that Convert the source code into object code
Object Code
Text written in machine language

- **Definition of Address Binding**
- **Address binding** is the process of mapping **program instructions and data (logical addresses)** to actual locations in **main memory (physical addresses)** so the CPU can access them during execution.
- Every program initially works with **logical addresses** (relative addresses), and the **Operating System + loader** translate them into **physical addresses**.
- ◆ **Types of Address Binding**
- **Compile-Time Binding**
  - If the memory location is known at **compile time**, the compiler generates **absolute addresses**.
  - Example: Old simple systems.
  - Limitation: If the starting address changes, the program must be **recompiled**.
- **Load-Time Binding**
  - Compiler generates **relocatable code** (logical addresses).
  - The actual binding to physical addresses happens when the program is **loaded into memory**.
  - More flexible — the same code can be loaded at different memory locations.
- **Execution-Time Binding**
  - The **CPU + MMU (Memory Management Unit)** translate logical addresses into physical addresses **dynamically during execution**.
  - Needed for **process relocation** (moving between RAM locations) and **virtual memory**.
  - Example: Modern operating systems (Linux, Windows).
- ◆ **Illustration**
- Imagine your C program has:
- int a = 10;
- **Compile-time binding**: Compiler fixes a at physical location 1000.
- **Load-time binding**: Compiler says "a is at offset 20". Loader decides actual memory (say 5000 + 20 = 5020).
- **Execution-time binding**: CPU generates logical address 20 → MMU maps it to physical address (5020, 7020, etc., depending on paging).

Address Binding

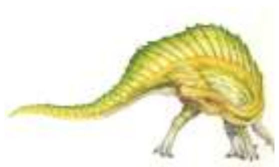
- Programs on disk, ready to be brought into memory to execute form an **input queue.**
- Although the address space of computer starts at 00000, first address of user process need not be 00000.
  - Source code addresses usually symbolic addresses
  - Compiled code will **bind** symbolic addresses to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
- address generated by the CPU is commonly referred to as a **logical address.**
- address seen by memory is commonly referred to as a **physical address**.



Binding of Instructions and Data to Memory

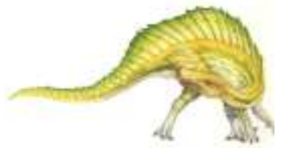
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: If memory location is not known a priori, compiler generates **relocatable code.** In this case final binding is delayed until load time.
  - **Execution time**: Binding delayed until run time.

  - In contrast to absolute code, which contains fixed memory addresses for the program's instructions and data, relocatable code uses relative memory addressing or symbolic references to access memory locations. This allows the code to be loaded into any memory location without requiring any modification to the code itself.

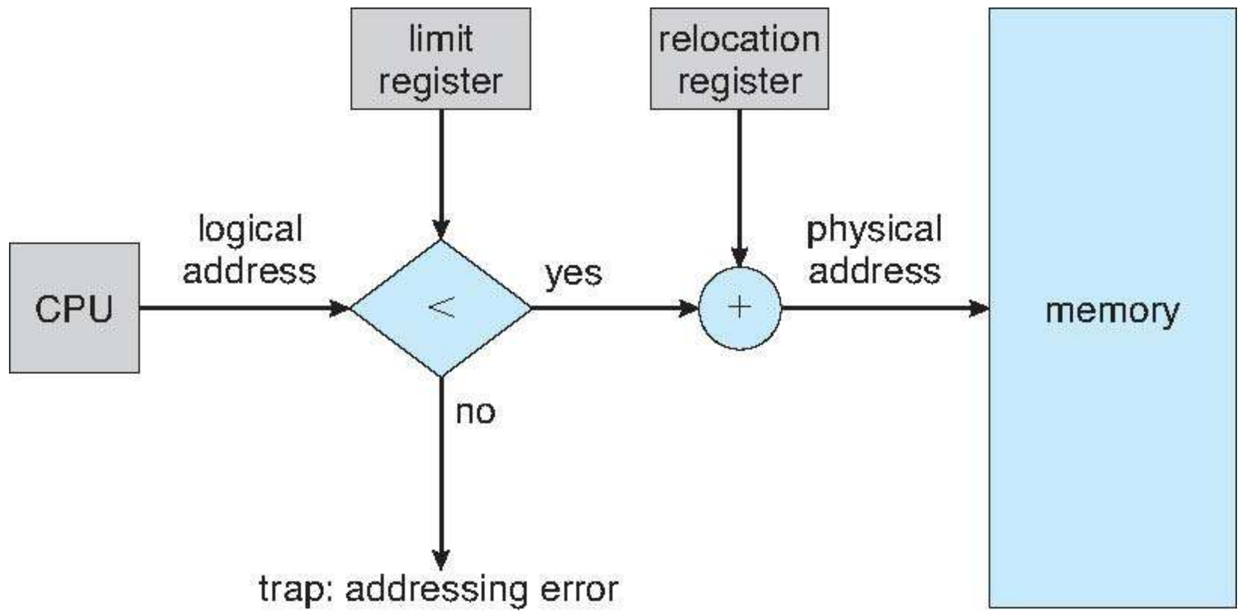


Logical vs. Physical Address Space

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



Hardware Support for Relocation and Limit Registers

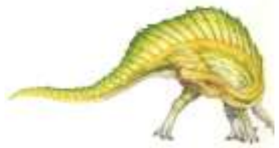


This protection must be provided by the hardware because the operating system doesn’t usually intervene between the CPU and its memory accesses (because of the resulting performance penalty).

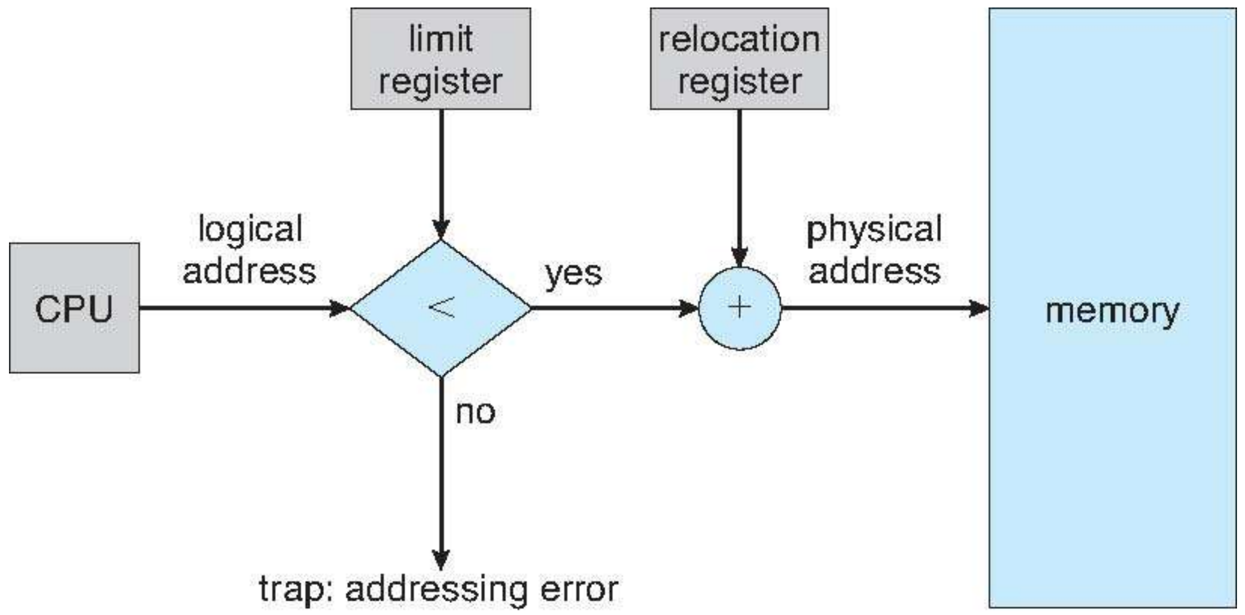
- The limit register stores the size of the process’s logical address space (the maximum number of addresses the process is allowed to generate).
This value is determined by the operating system at the time the process is loaded into memory. When a program is compiled, the compiler/assembler produces object code with information about the size of its code, data, and stack segments.

Memory-Management Unit (MMU)

- Hardware that at run time maps virtual to physical address
- Many methods possible
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - **8086 used 4 relocation registers**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- mapping from virtual to physical addresses is done by a hardware called the MMU



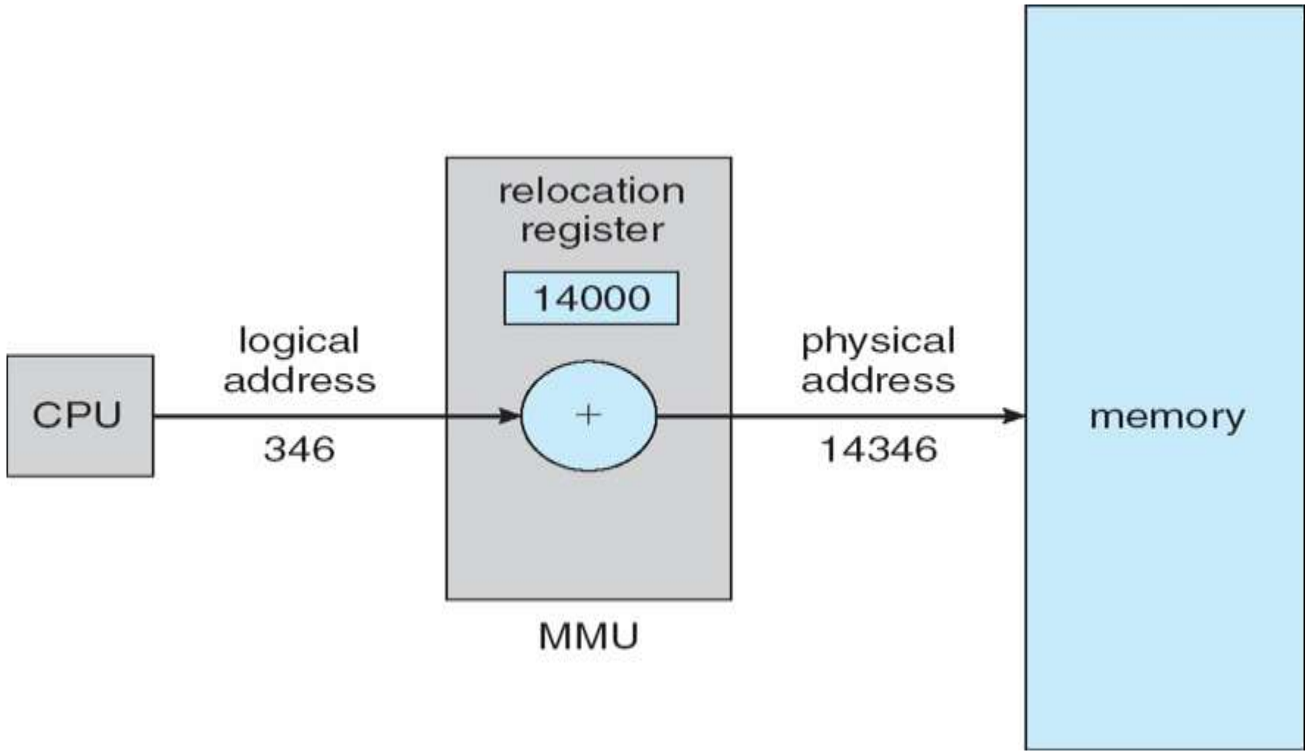
Hardware Support for Relocation and Limit Registers



Dynamic relocation using a relocation register

- **<u>A simple MMU mapping scheme</u>**

•Relocation Register = ensures the program can be loaded anywhere in memory..
•Limit Register = handles protection (ensures process doesn’t go out of bounds).



Dynamic Loading

- **Entire program and all data need not be in memory for execution**
- For better memory utilization; use dynamic loading
- With dynamic loading, **a routine is not loaded until it is called.**
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking- loader is called to load the desired routine into memory.
- The advantage of dynamic loading is that an unused routine is never loaded.

Relocatable linking- loader: is a type of loader that combines the functions of both linking and loading

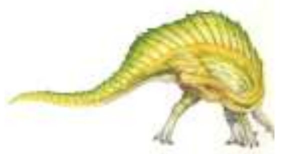
The relocatable linking-loader adjusts the memory addresses in the program so that it can be correctly executed from whatever memory location it is loaded into

Dynamic Linking

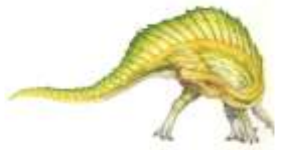
- **Static linking** – system libraries and program code combined by the linker into the binary program image before execution
- Dynamic linking –linking postponed until execution time
- Instead of embedding full library code in the executable, the program contains a **small piece of code called a stub(**runtime, dynamic, placeholder code → *used to connect to real code later*.**)**
- The stub helps locate the required library routine at runtime.
- If the routine is not already loaded in memory, the operating system loads it.
- The stub then replaces itself with the address of the library routine and executes it.
- Dynamic linking is particularly useful for libraries

gcc -o my_program my_program.c -static

gcc -o my_program my_program.c –lm (link dynamically the math library)

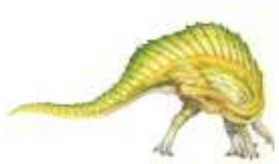


- A **stub** is a small piece of code that acts as a **placeholder** or **intermediary** between your program and the actual routine in a shared library.
- When your program calls a library function (say printf()), it doesn't jump directly to the function in the shared library.
- Instead, it first goes through the **stub**.
- The stub's job is to:
  - **Check if the routine is already loaded and linked.**
  - If not, **load/link the routine at runtime** (via the dynamic linker/loader).
  - Pass control to the actual routine once resolved.

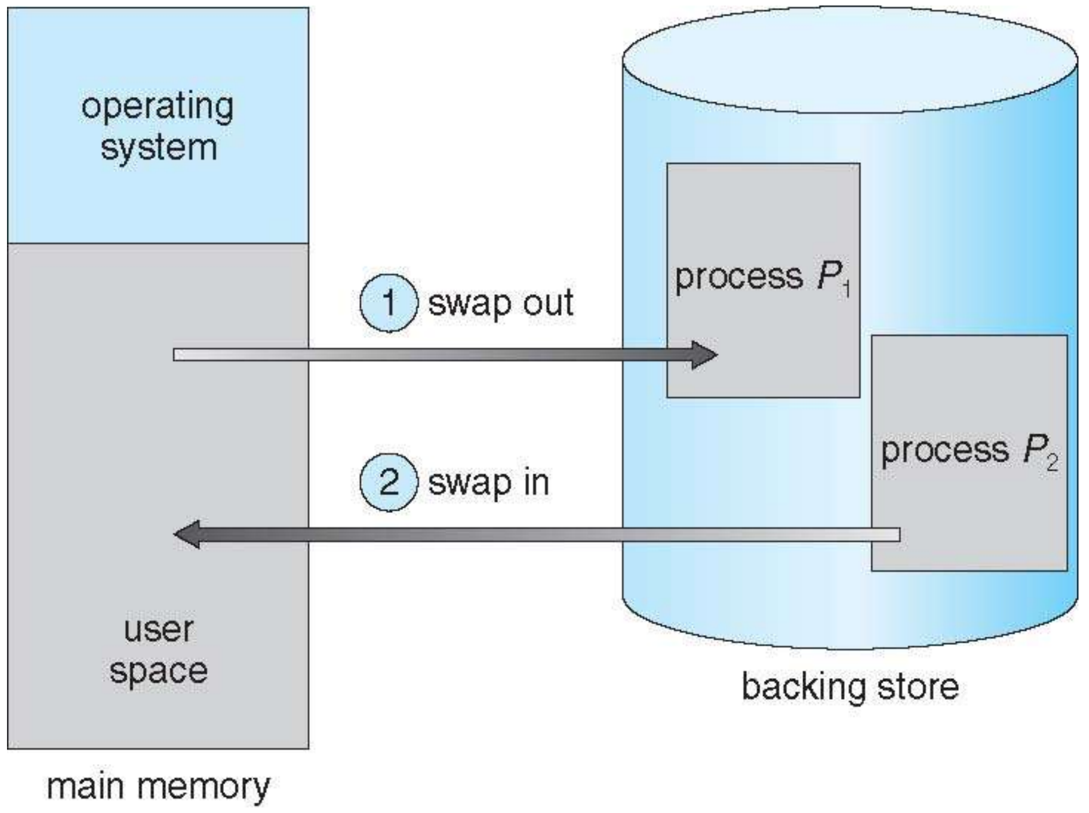


Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

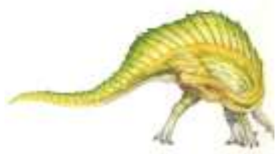


Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2seconds
  - 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()—requests additional memory` and `release_memory()—releases the memory that is no longer 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
  - Each process contained in single contiguous section of memory

Today's general-purpose OSes (Windows, Linux, macOS) do not use contiguous allocation for processes because of its limitations:
•External fragmentation wastes memory.
•Fixed partitions limit flexibility.
•Dynamic resizing of processes is difficult.
•Multiprogramming is less efficient

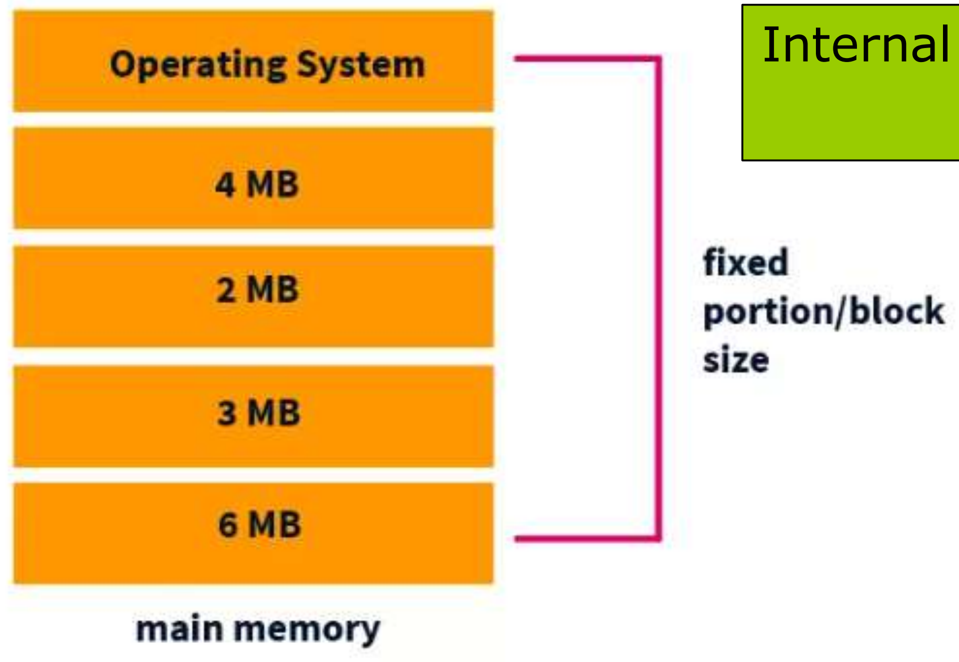
Multiple-partition allocation

Multiple-partition allocation

- **Fixed-partition**
- Memory is divided into a **fixed number of partitions** at system startup
- Each partition can hold exactly one process.
- Degree of multiprogramming = limited by the number of partitions.
- If process is smaller than its partition → internal fragmentation (unused space inside partition). If process is larger than any partition → it cannot be loaded.
- **Variable-partition**
- Initially, all memory is available for user processes →one large block of available memory
- we have a list of available block sizes and an input queue(set of processes that enter the system).
- Memory is allocated to processes until finally, the memory requirements of the next process cannot be satisfied
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one large hole.
- However, as processes load and exit, free spaces (called holes) get scattered → leads to external fragmentation.

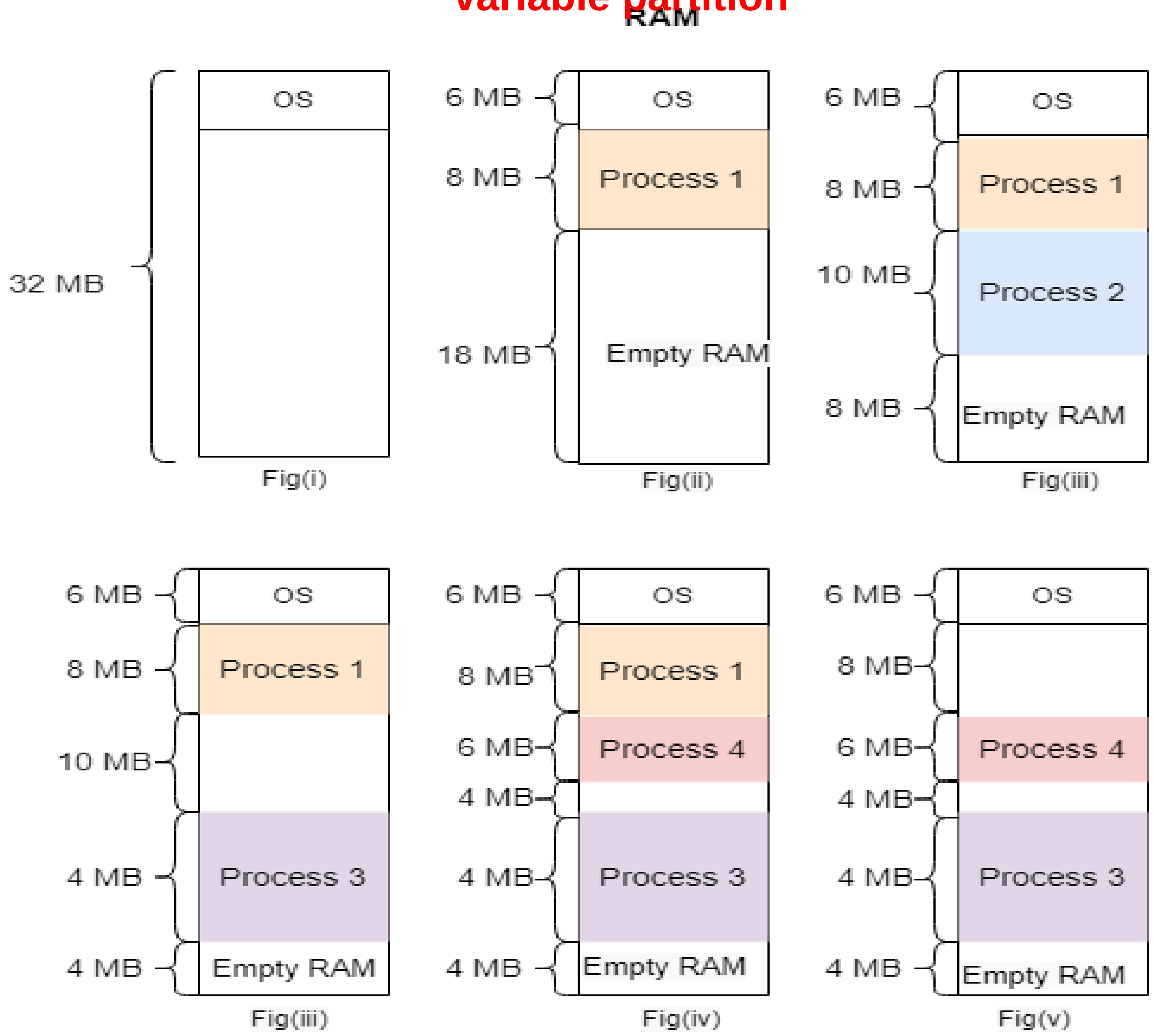
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)

fixed partition ➔ fixed number of partitions



Internal fragmentation

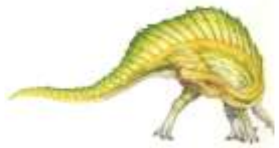
variable partition



1)NO Internal Fragmentation
2)There exists External Fragmentation
3)Uses compactio

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous(Variable sized memory)
- **Internal Fragmentation** – available memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used(Fixed size memory)
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block

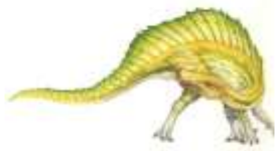


Dynamic Storage-Allocation Problem

Allocation Strategies (for Variable-Partition Systems)

- **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



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

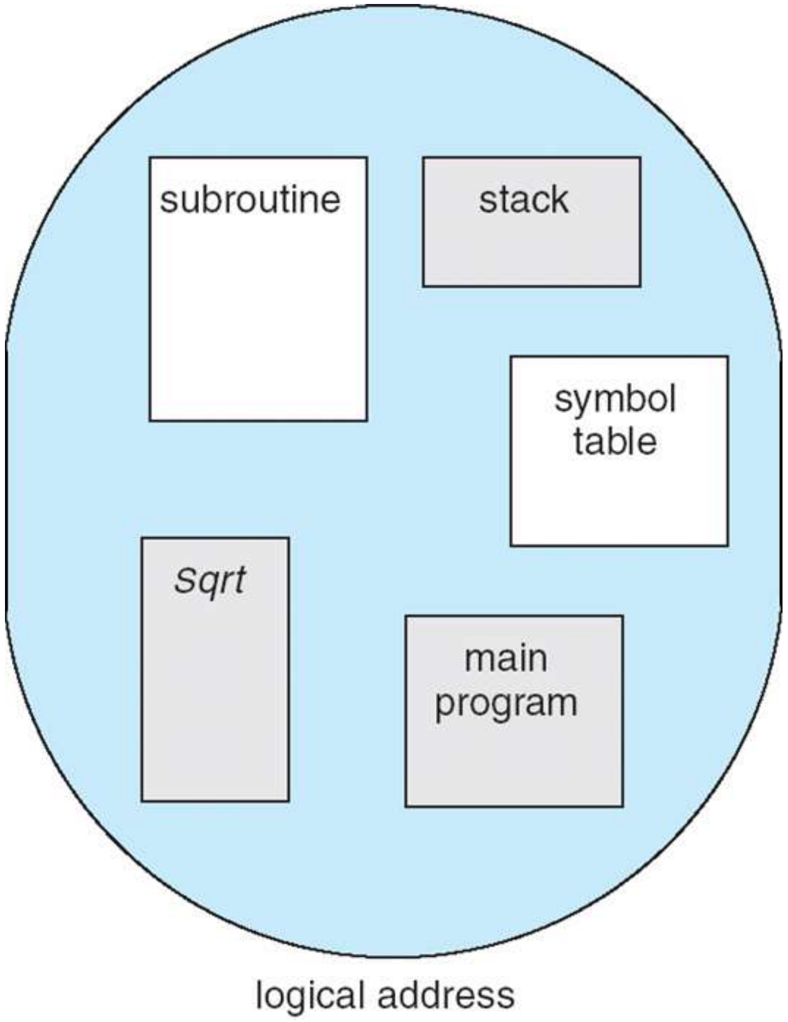
    common block

    stack

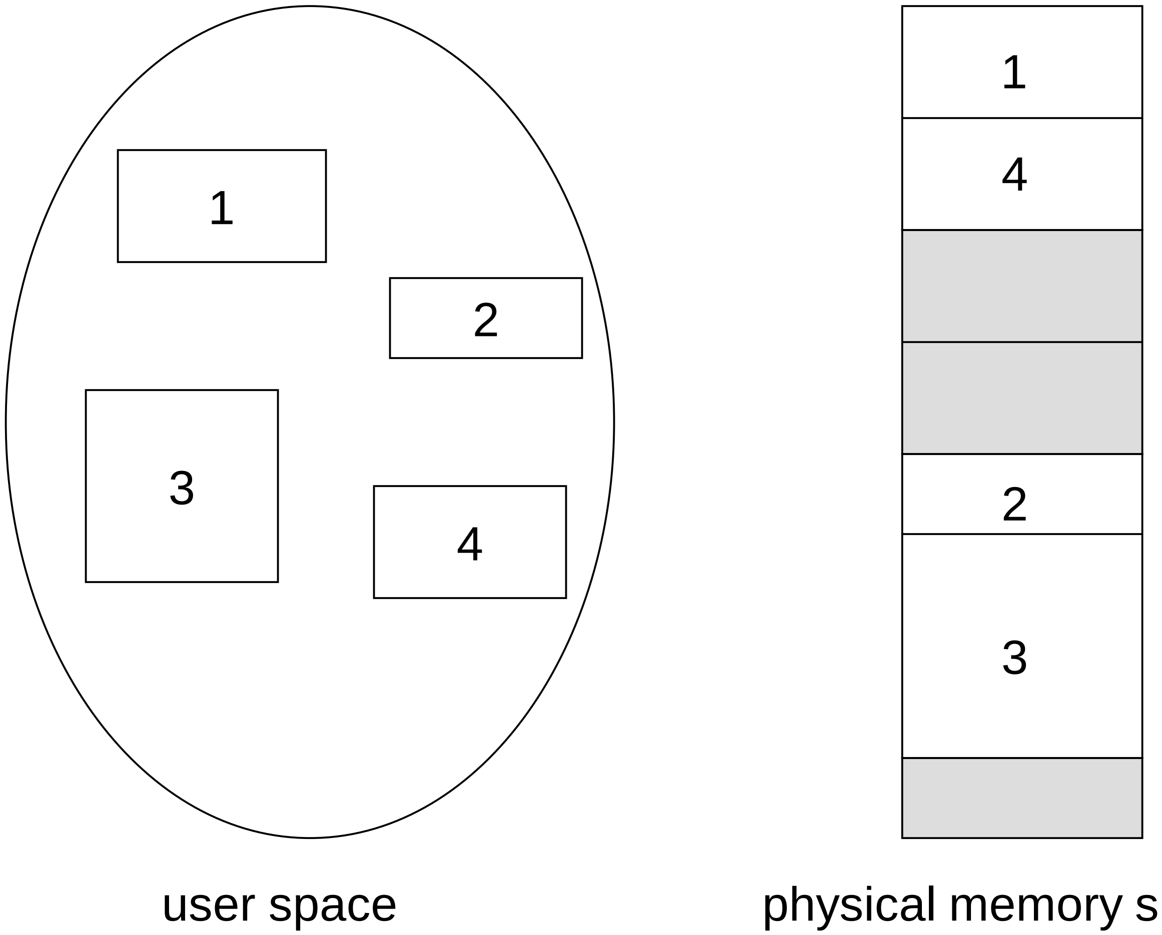
    symbol table

    arrays

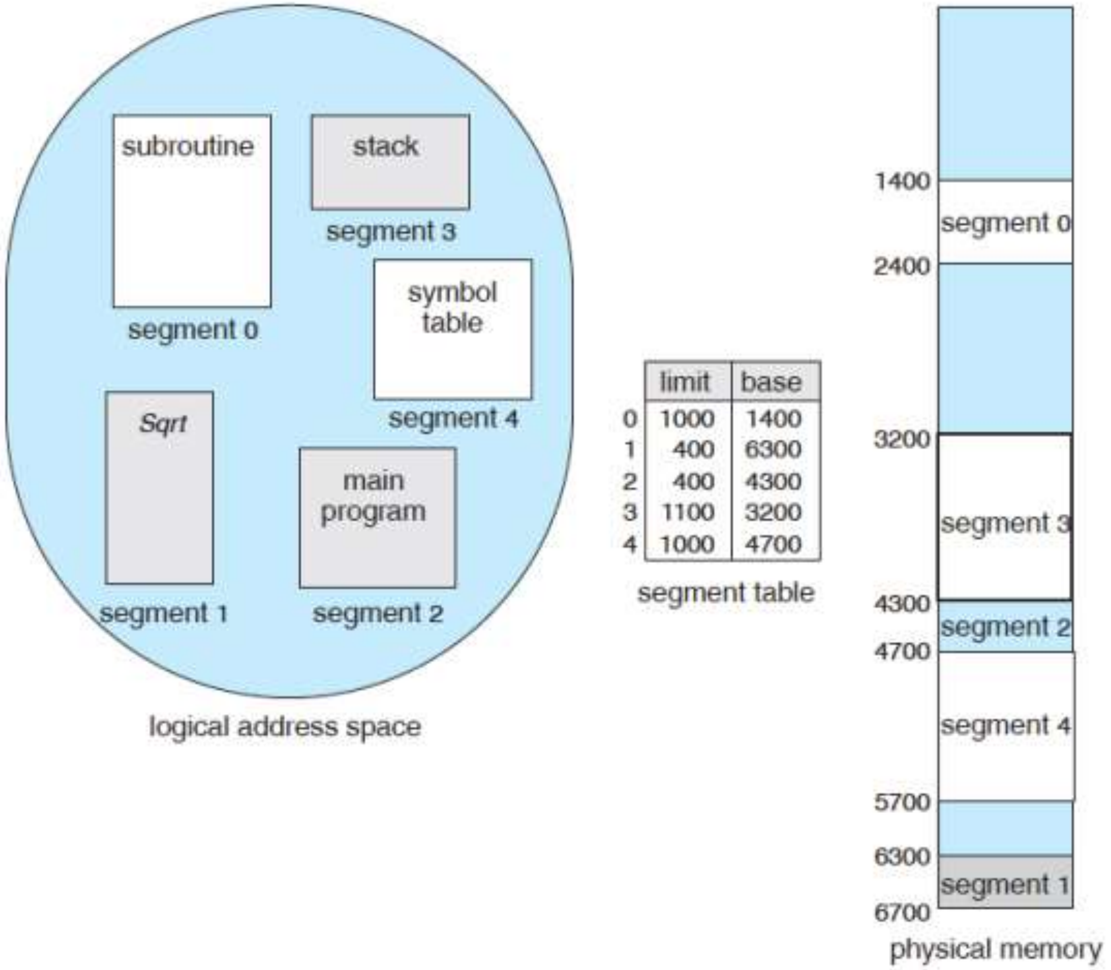
User' s View of a Program



Logical View of Segmentation



Example of segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

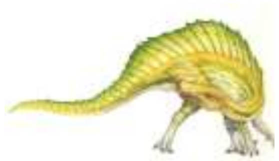
segment table

Segmentation Architecture

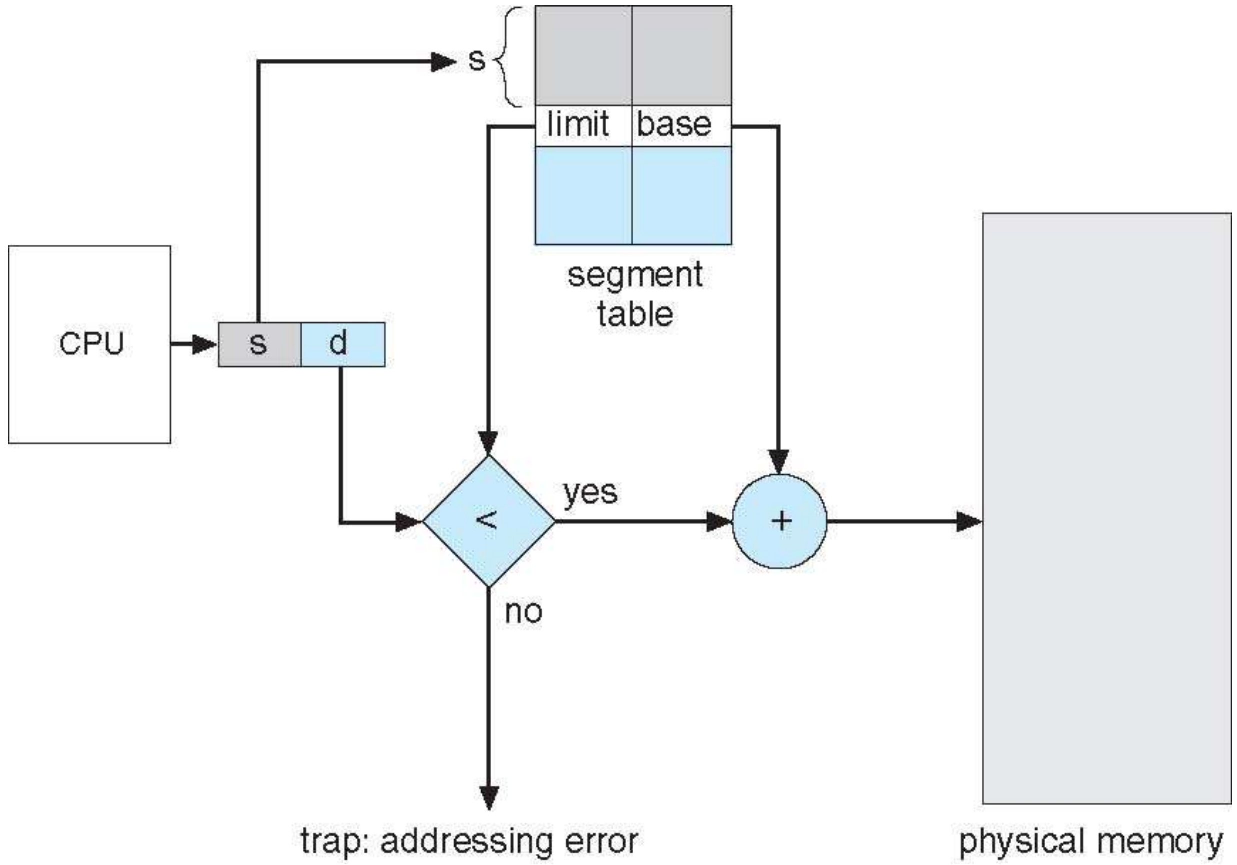
- Logical address consists of a two tuple:

  <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
- **The segment table is thus essentially an array of base-limit register pairs.**
- **Segment-table base register (STBR)**
- **Segment-table length register (STLR)**

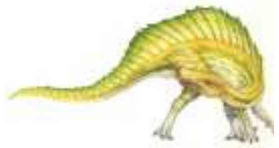


Segmentation Hardware

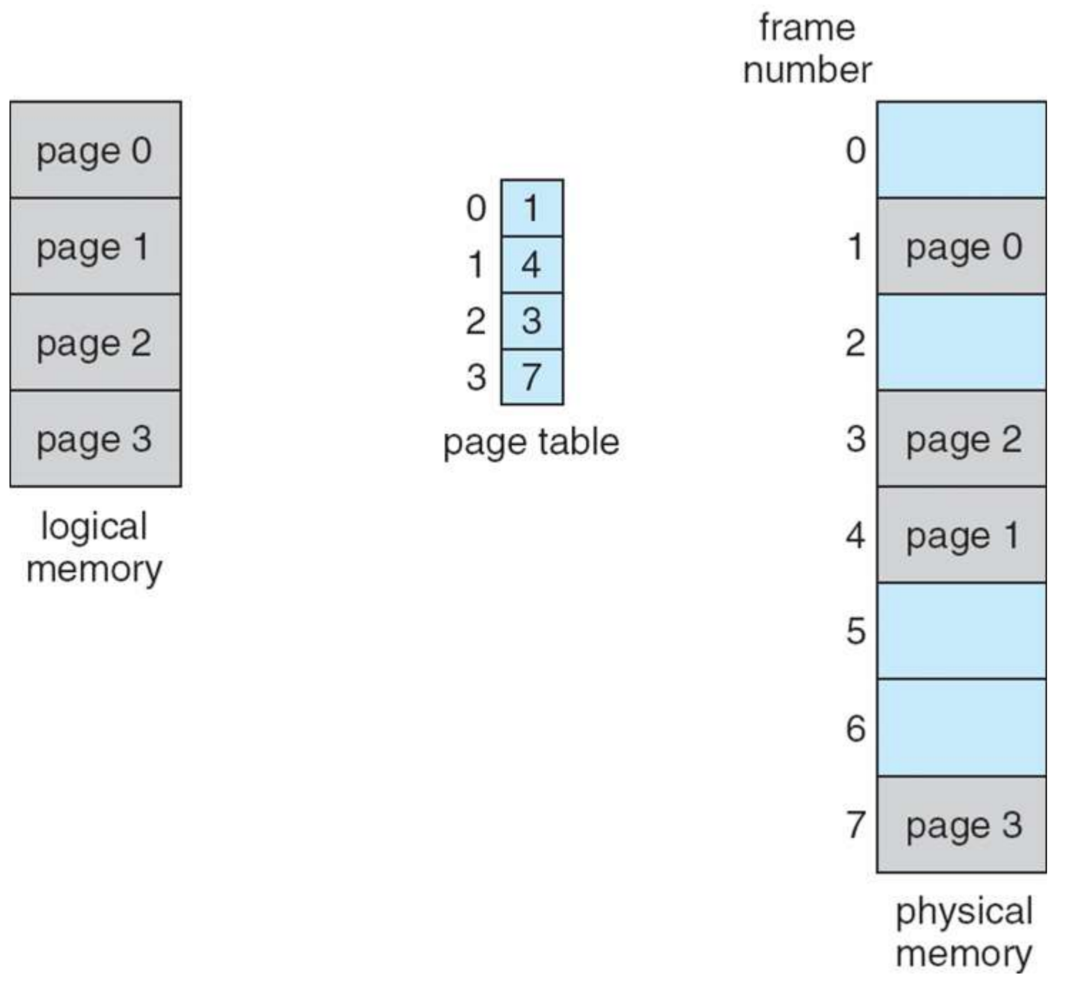


Paging

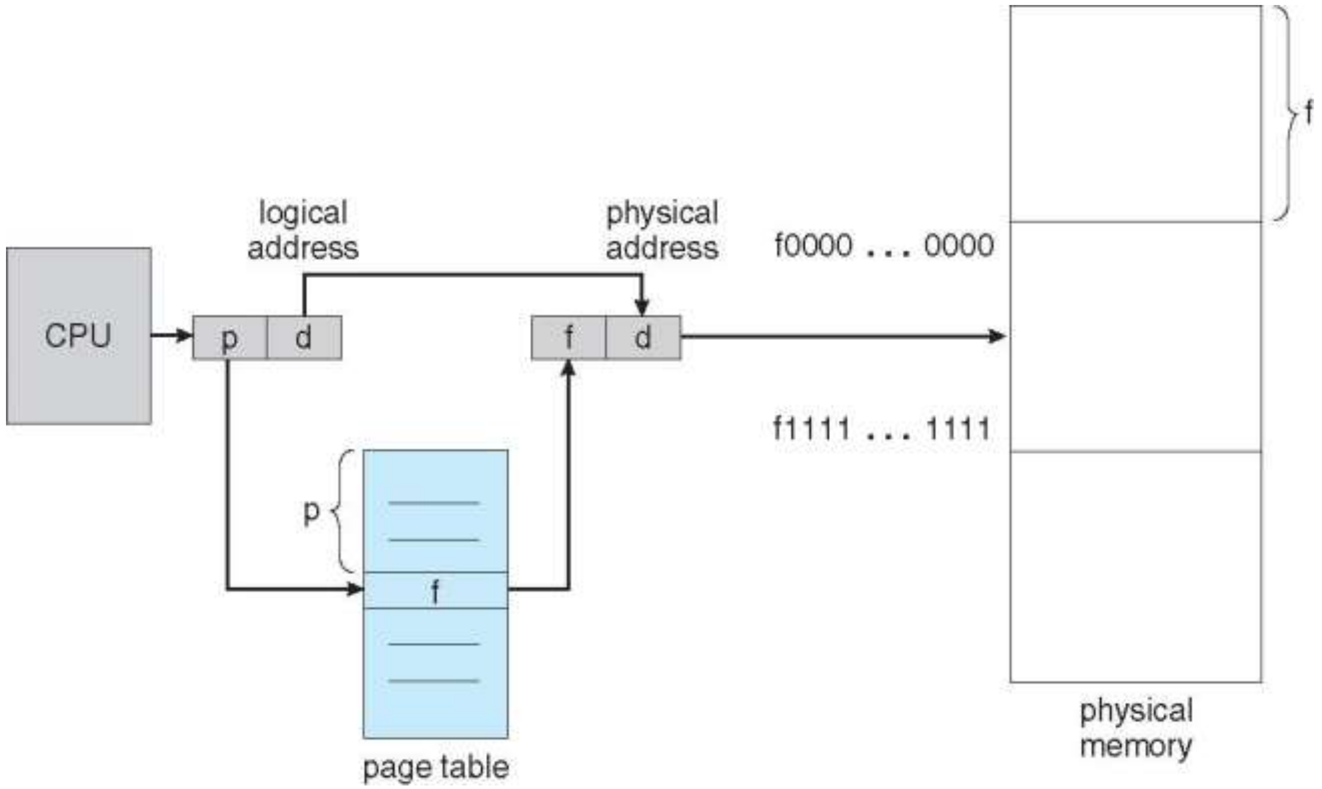
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
- **Page size=frame size**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses



Paging Model of Logical and Physical Memory

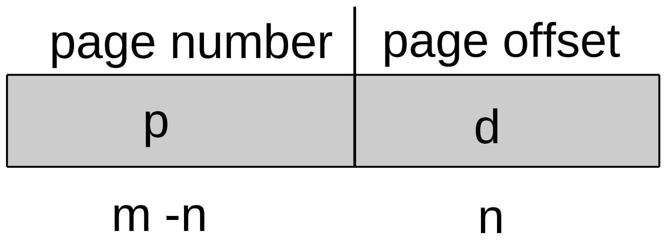


Paging Hardware

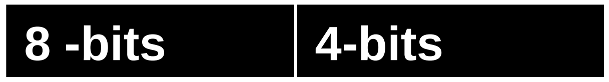


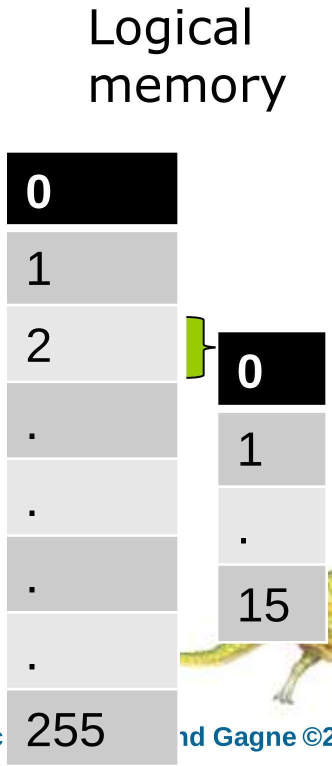
Address Translation Scheme

- 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

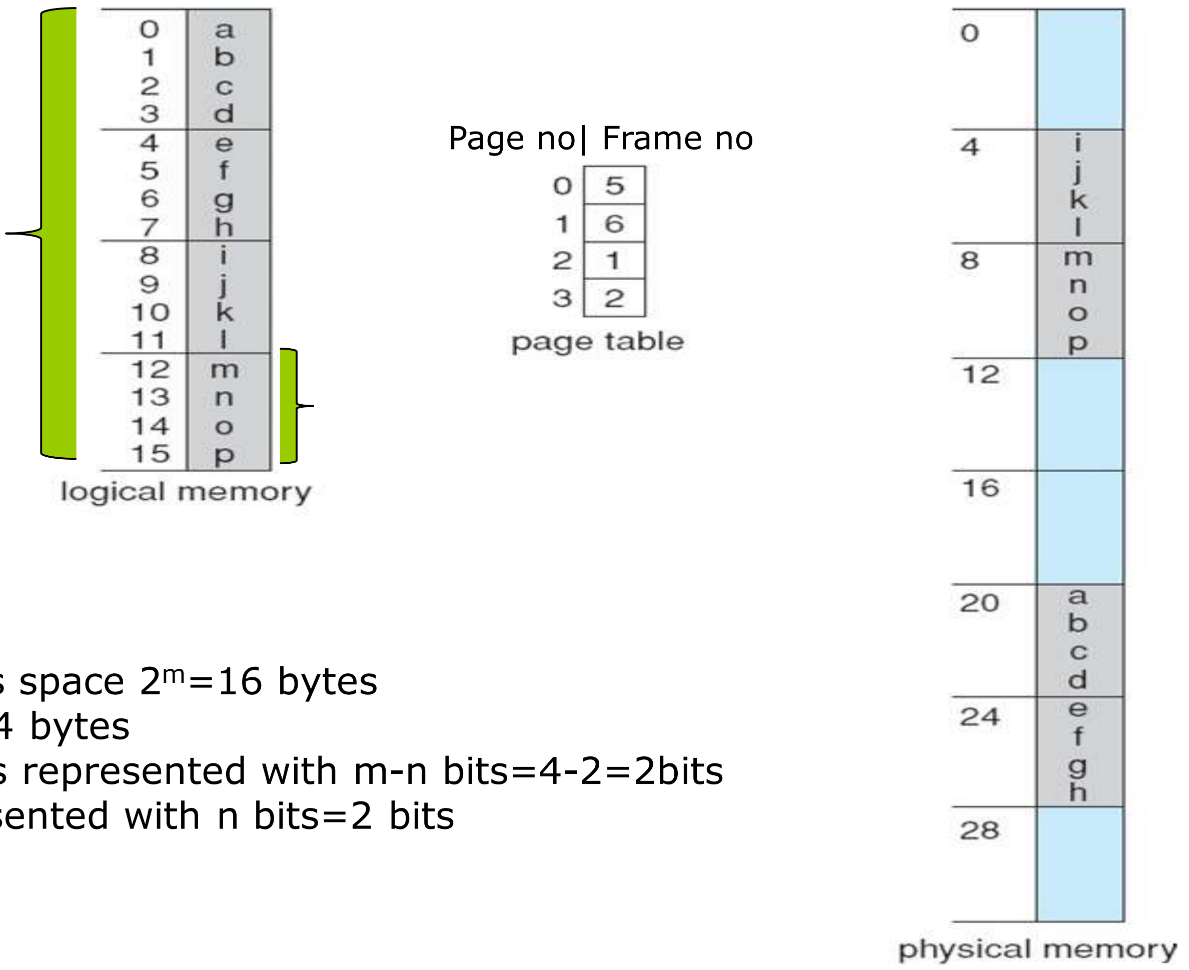




Paging Example

n=2 and *m*=4

32-byte main memory and 4-byte pages



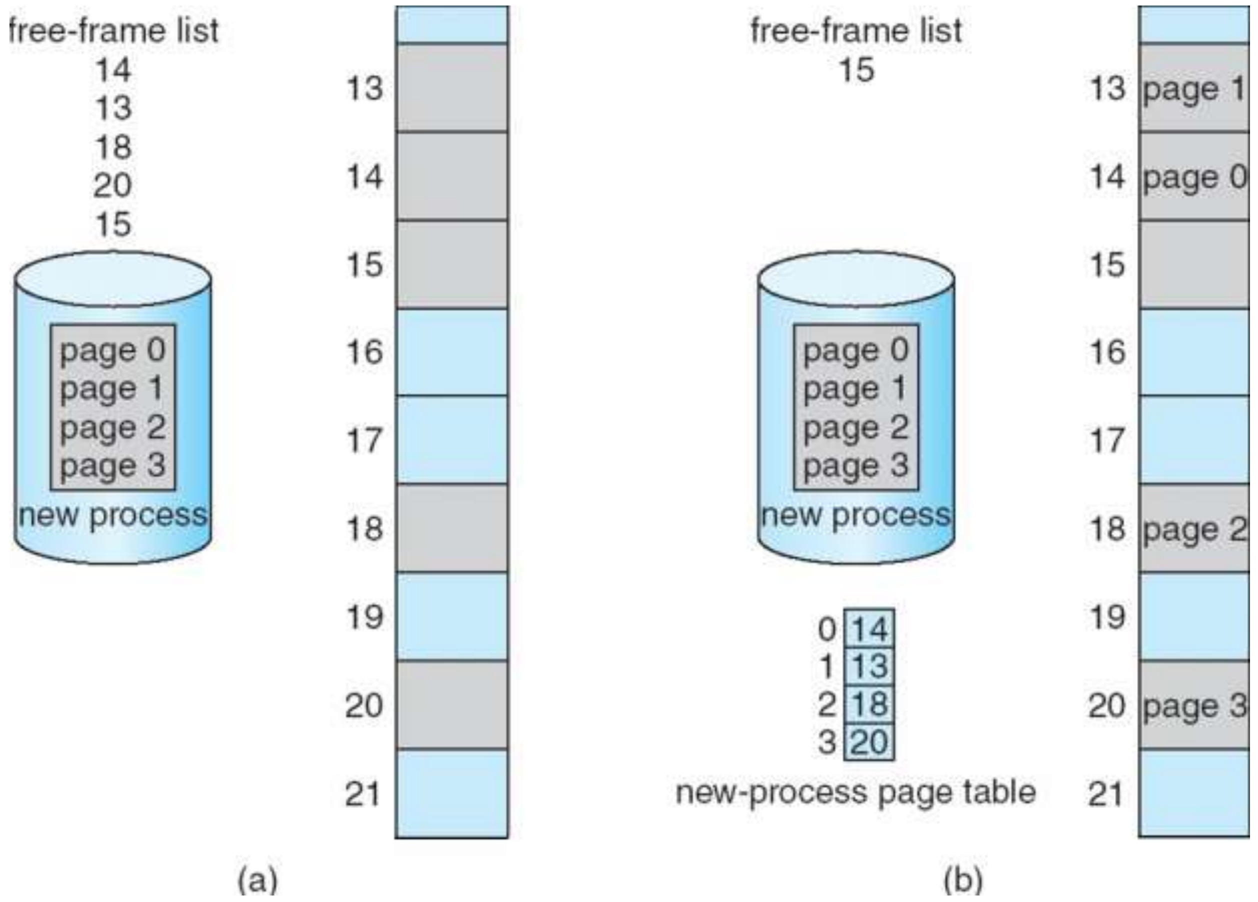
Page no	Frame no
0	5
1	6
2	1
3	2

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

Logical address space 2^m=16 bytes
Page size=2^n=4 bytes
Page number is represented with m-n bits=4-2=2bits
Offset is represented with n bits=2 bits

Free Frames

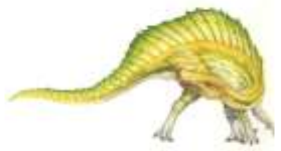


Before allocation

After allocation

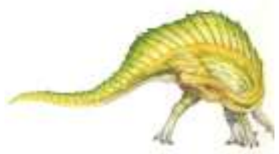
Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes   (35x2048=71680)+1086
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte



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 present in the physical memory frame.
- 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**)



Associative Memory

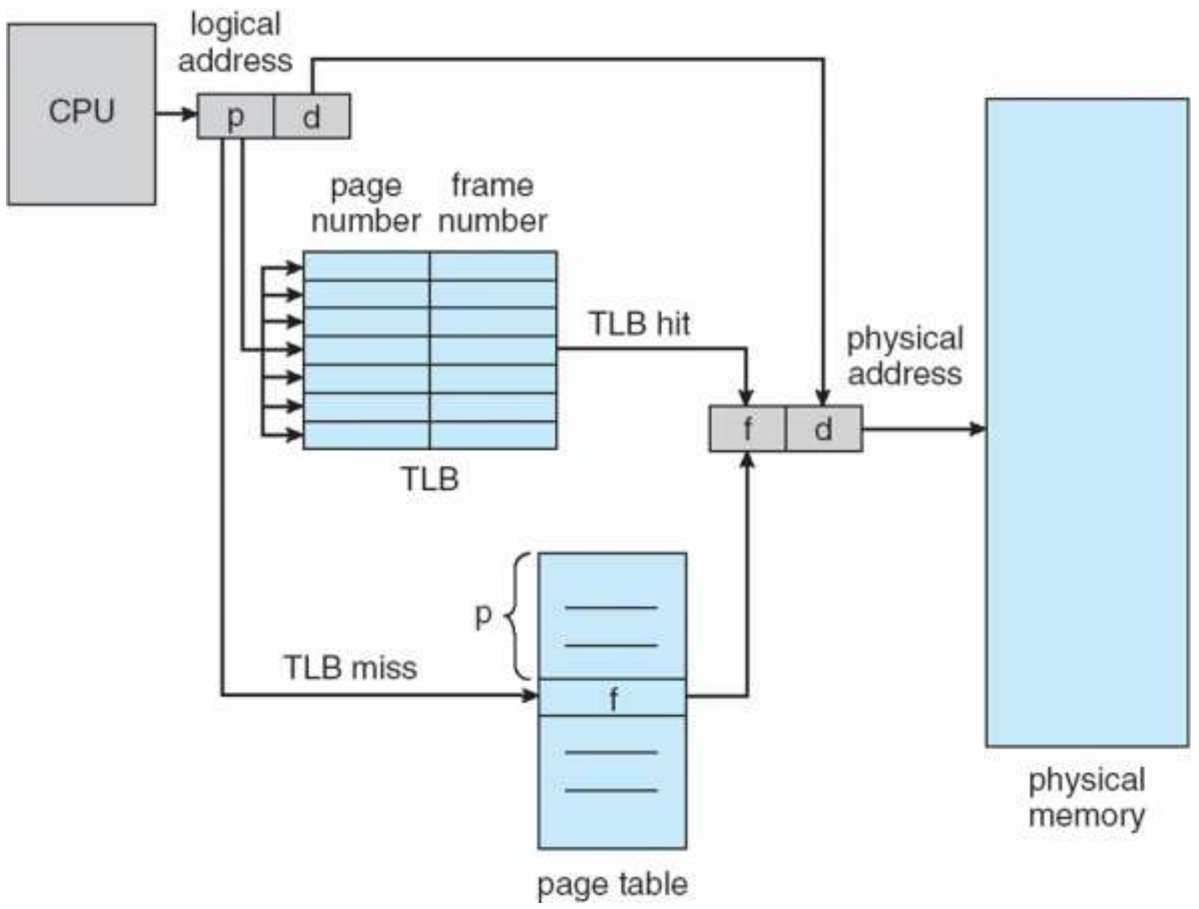
- On a TLB miss, value is loaded into the TLB for faster access next time

- 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 associative registers
- TLB hit 80%, 20ns for TLB search, 100ns for memory access, then

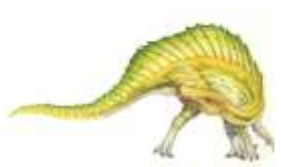
  a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.

- If we fail to find the page number in the TLB (20 nanoseconds),then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds),for a total of 220 nanoseconds
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 120 + 0.20 x 220 = 140ns

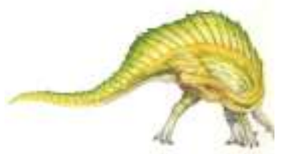
For 98% hit ratio

EAT = 0.98 x 120 + 0.02 x 220 = 122ns



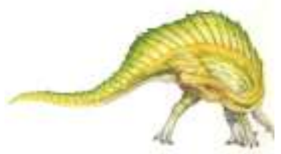
- A process’s logical address space might be much smaller than the maximum space supported by hardware.
- Example: hardware may allow 16 pages, but the program only needs 9.
- Without V/I bit, the page table entries for the unused pages would still exist → the CPU might wrongly try to use them.
- With V/I bit, those entries are marked **invalid**.

- Prevents a process from accessing memory outside its legal logical space.
- If CPU generates an address to an invalid page → page table entry says invalid → OS trap.

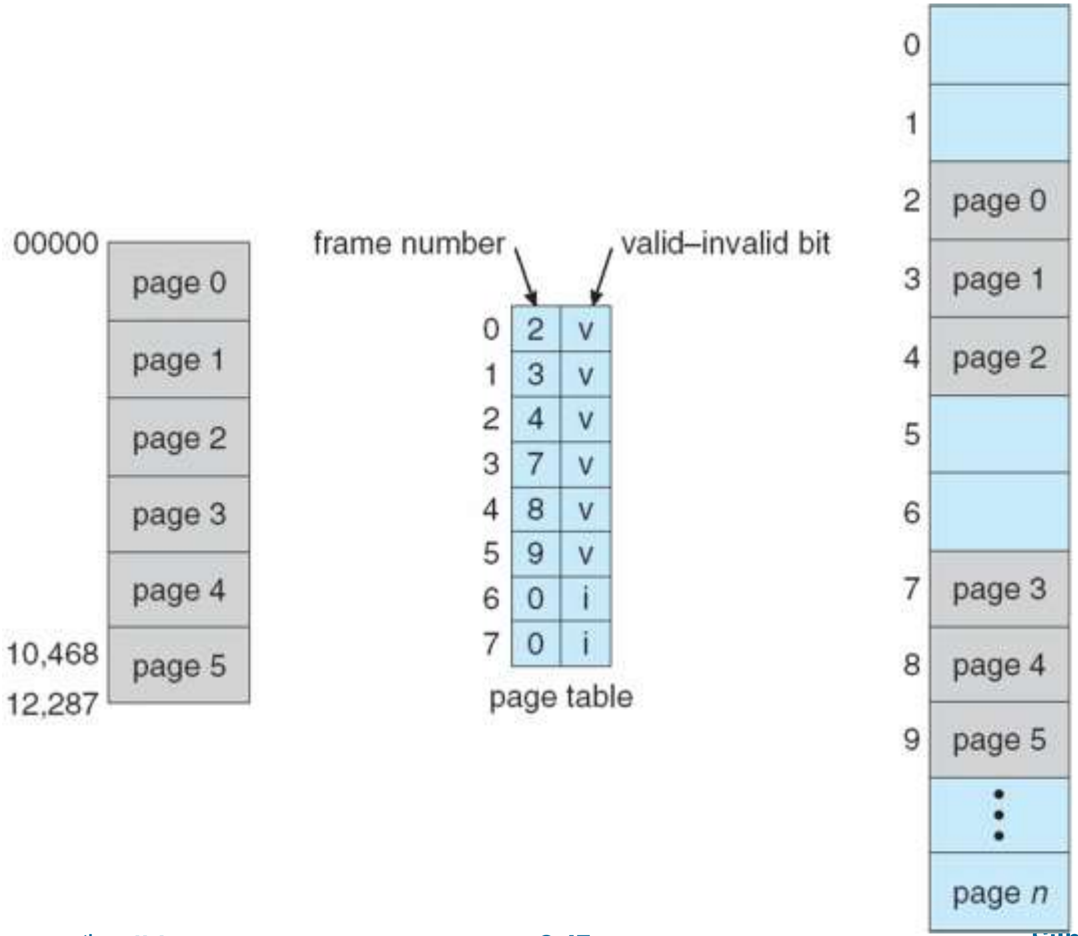


Memory Protection

- Memory protection implemented by associating protection bit.
- **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**) for memory protection.
- Any violations result in a trap to the kernel



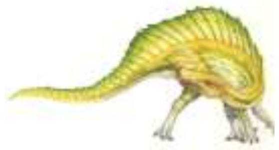
Valid (v) or Invalid (i) Bit In A Page Table



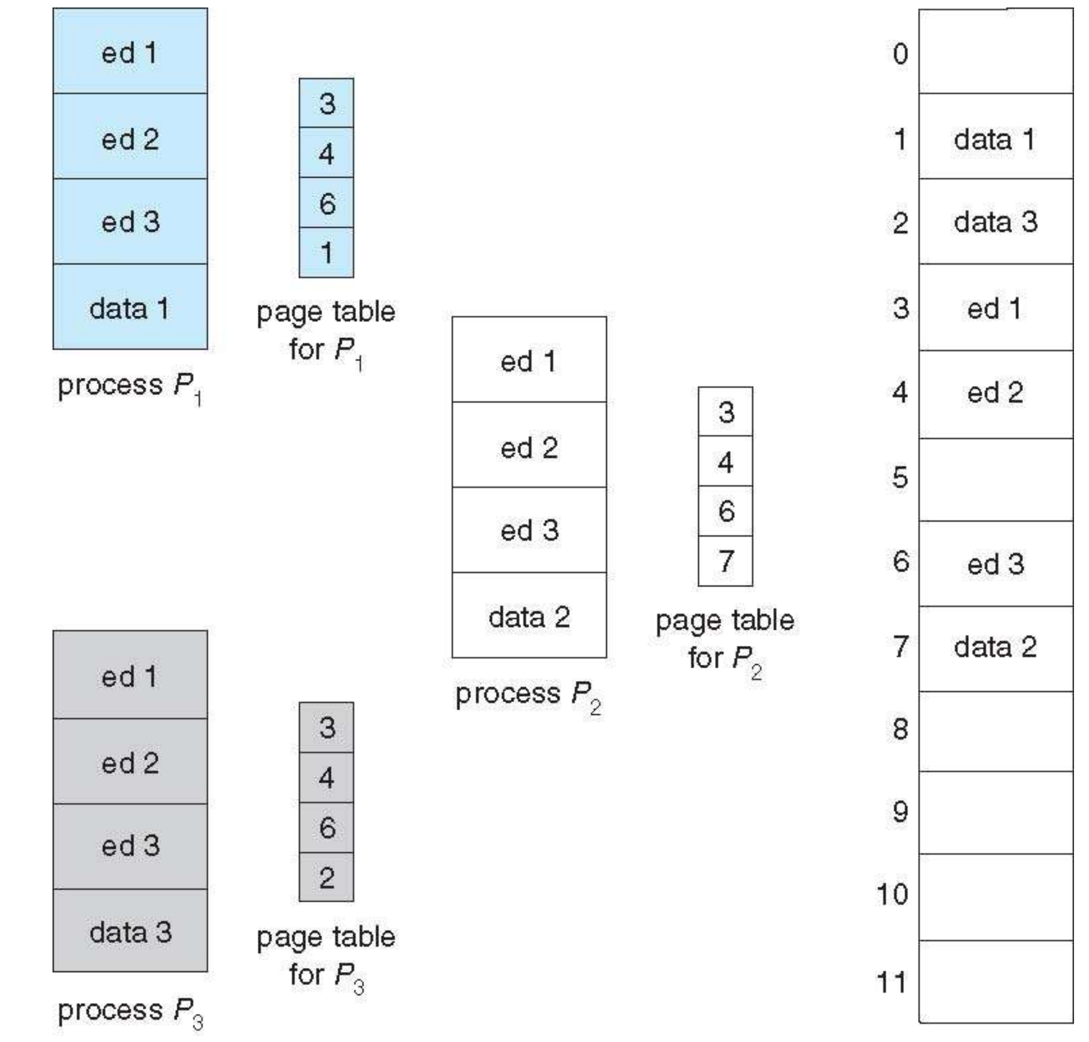
Shared Pages

- In a pure paging system, each process has its own page table.
- If two processes run the same program, the OS loads separate copies of that program’s code into different physical frames.
- Problem: wastes memory, because the code is identical for all processes.
- 10 users run the same compiler. Without shared pages → 10 copies of the compiler’s code in memory.
- **Shared code**
- Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers etc)
  - Similar to multiple threads sharing the same process space
- **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

- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB(150*40+50*40) to support the 40 users.
- To support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2150 KB(150+50*40) instead of 8,000 KB
- Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages



Shared Pages Example



End of Chapter 8