Chapter 8: Memory Management

NARZU TARNNUM
CSE, BU

Chapter 8: Memory Management

- Background
- Swapping
- Memory Allocation
- Paging
- Segmentation

Memory Management

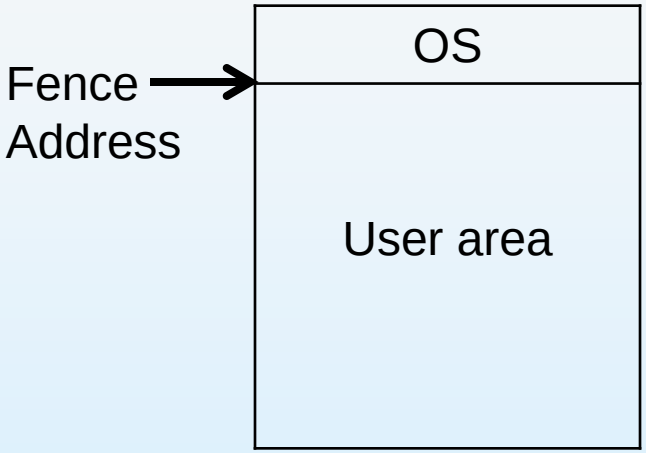
- Main memory and registers built into the processor itself are the only storage that the CPU can access directly.
- It is the functionality of an operating system which manages primary memory and moves processes back and forth between main memory and disk during execution. Goal is to ensure proper utilization of space and run long process using smaller area.
- Keep track of every memory location. Take decision which process will get how much memory and when.
- Update status of memory location: allocated or free.

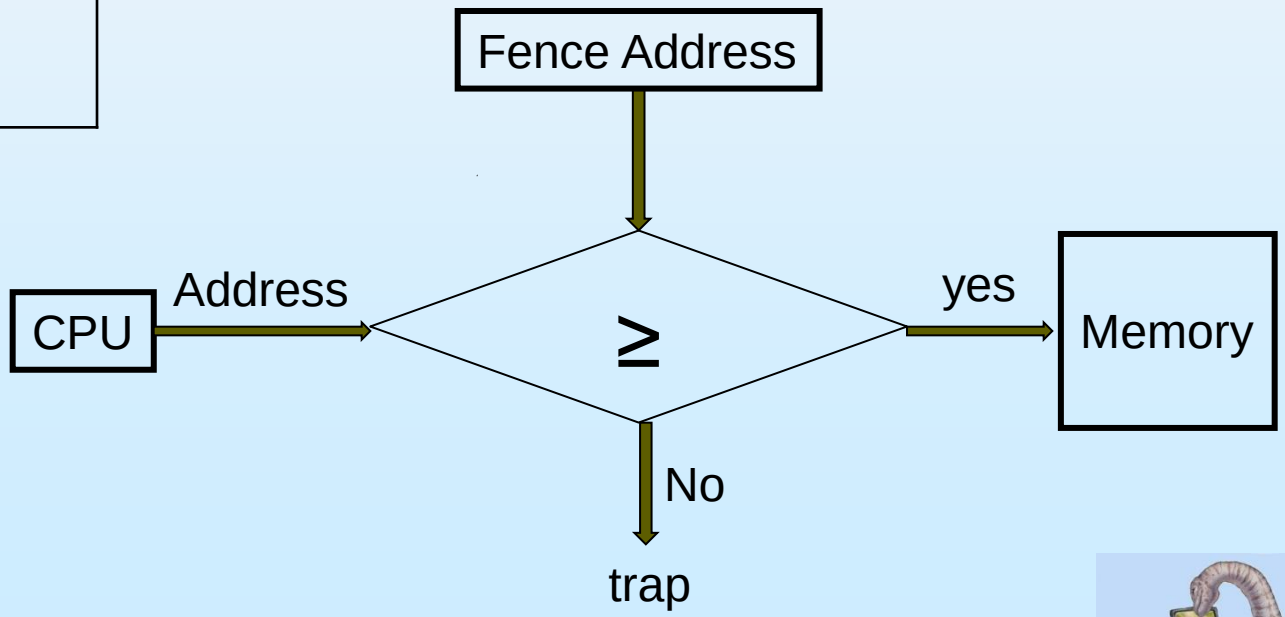
Protection

- In resident monitor memory model, there are two parts of memory:
  - Operating System area
  - User area
- One process can't access OS and other process's memory location.

Hardware comparator

- ✓ Fence Address
- ✓ Fence Register



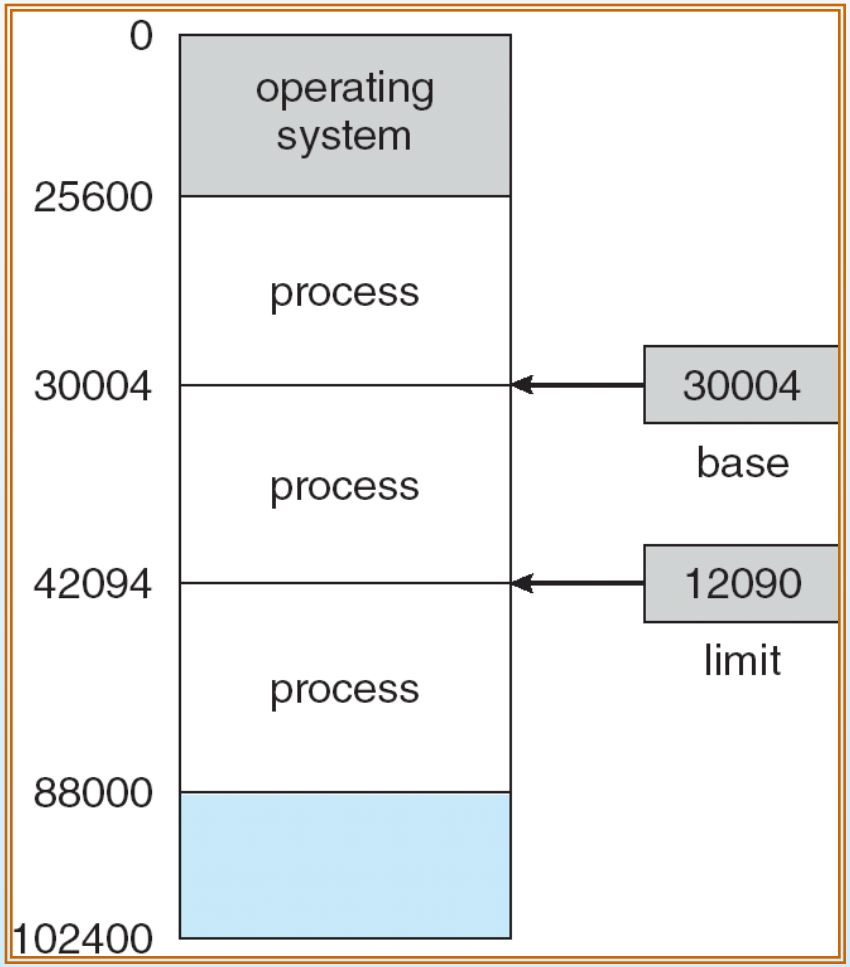


Basic Hardware

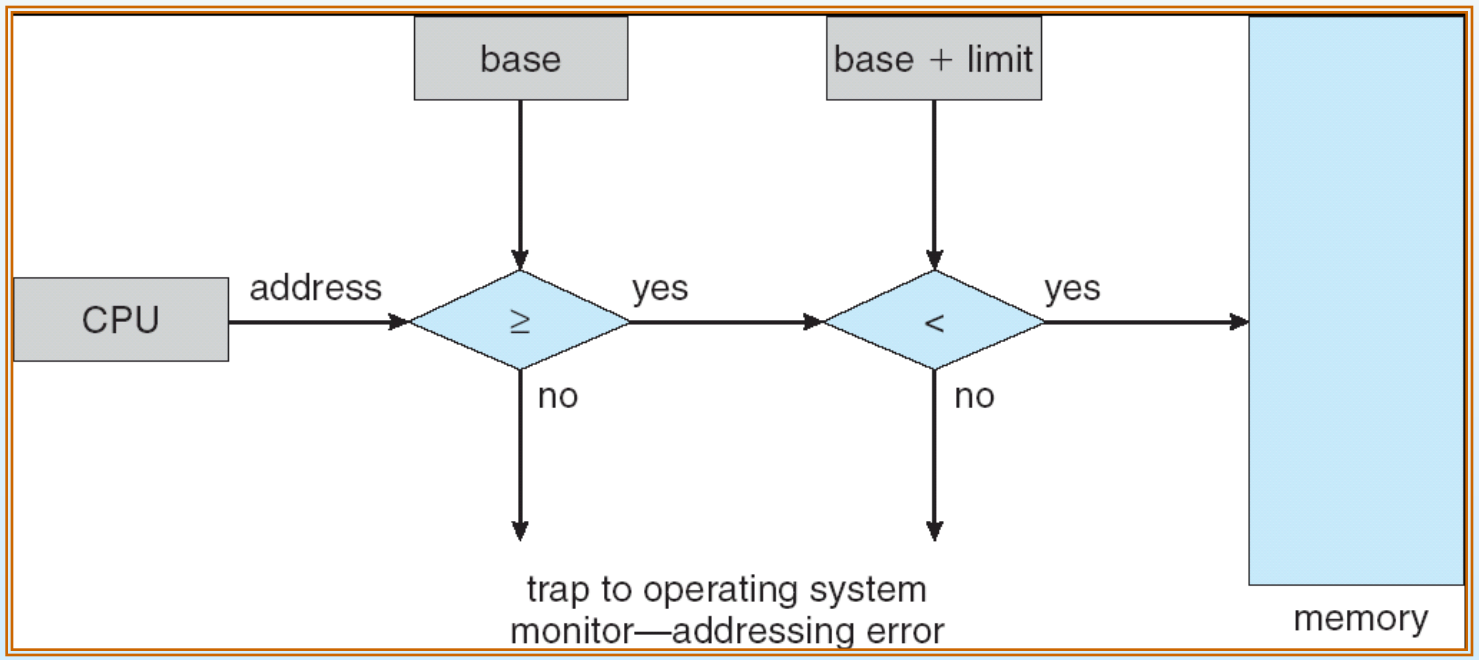
- Program must be brought into memory and placed within a process for it to be run

- Make sure that each process has a separate memory space which ensure the legal address spaces. Two registers are used to provide protection:
  - **Base register** holds smallest legal physical memory address
  - **Limit register** holds size of the range

A base and a limit register define a logical address space



HW address protection with base and limit registers



Address Binding

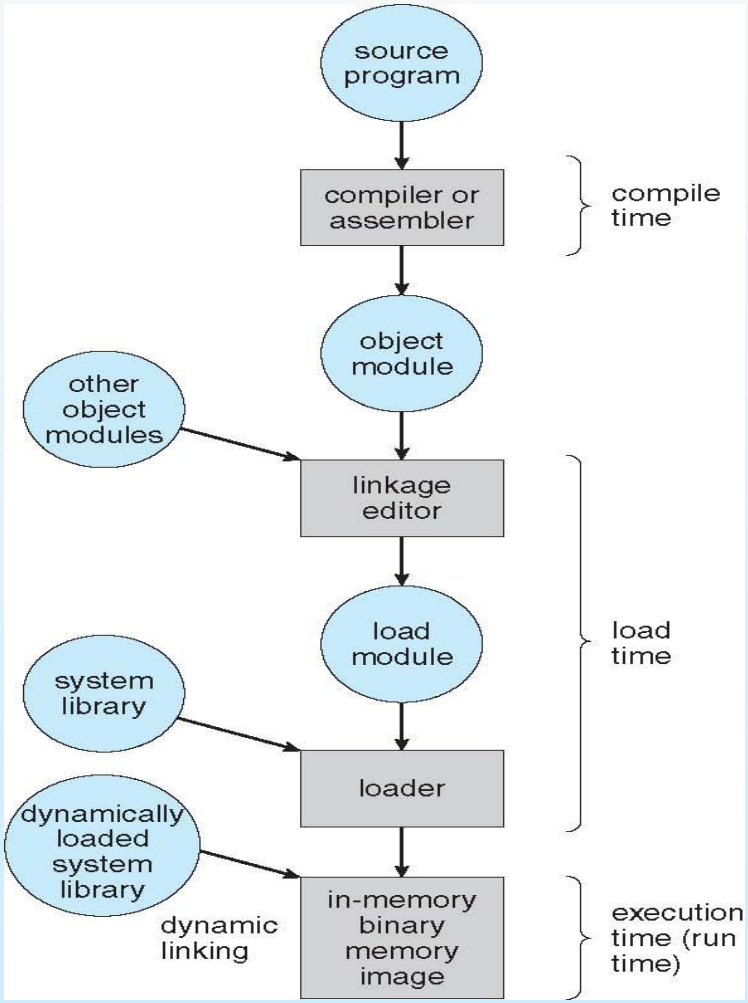
- The processes on the disk that are waiting to be brought into memory for execution from the **Input queue** .
- As the process is executed, it accesses instructions and data from memory.
- When the process terminates, its memory space is declared available.
- Address in the source program are generally symbolic.
- A compiler bind these symbolic addresses to re-locatable addresses.
- The linkage editor or loader will in turn bind the re-locatable addresses to absolute addresses.
- Each binding is a mapping from one addreess 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 of address Binding



Logical vs. Physical Address Space

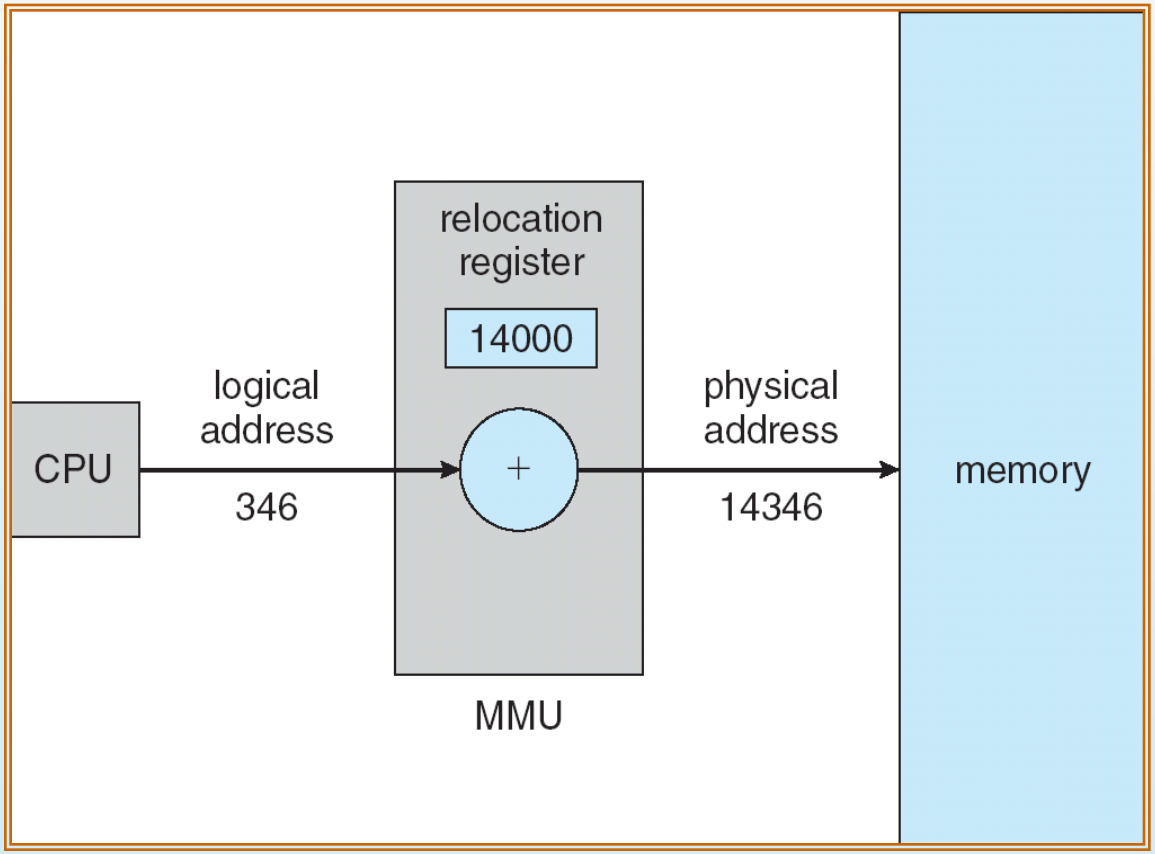
- **Logical address** – generated by the CPU; also referred to as *virtual address*
- **Physical address** – address seen by the memory unit

- 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)

- The run-time mapping from virtual to physical addresses is done by a hardware device called memory management unit.
- 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

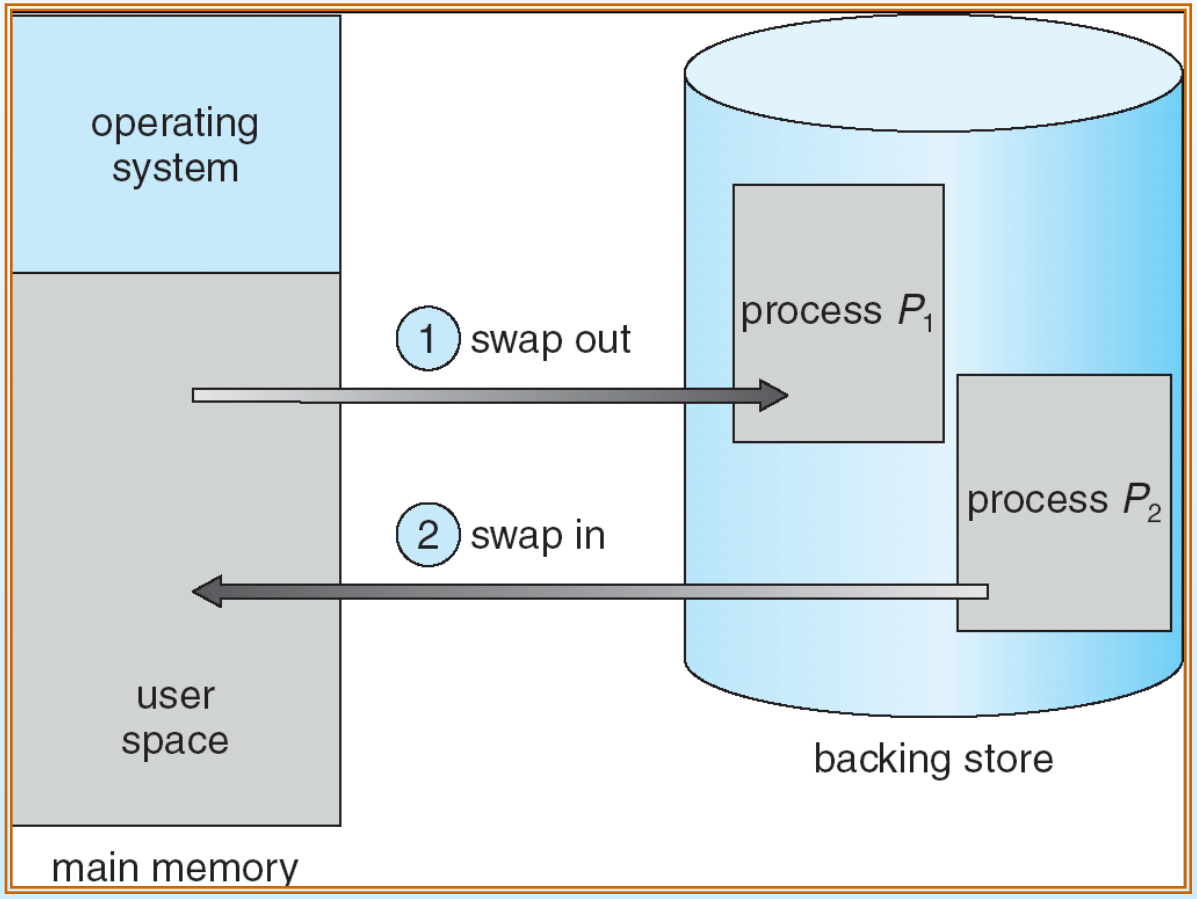
Dynamic relocation using a relocation register



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)

Schematic View of Swapping



Memory Management Technique

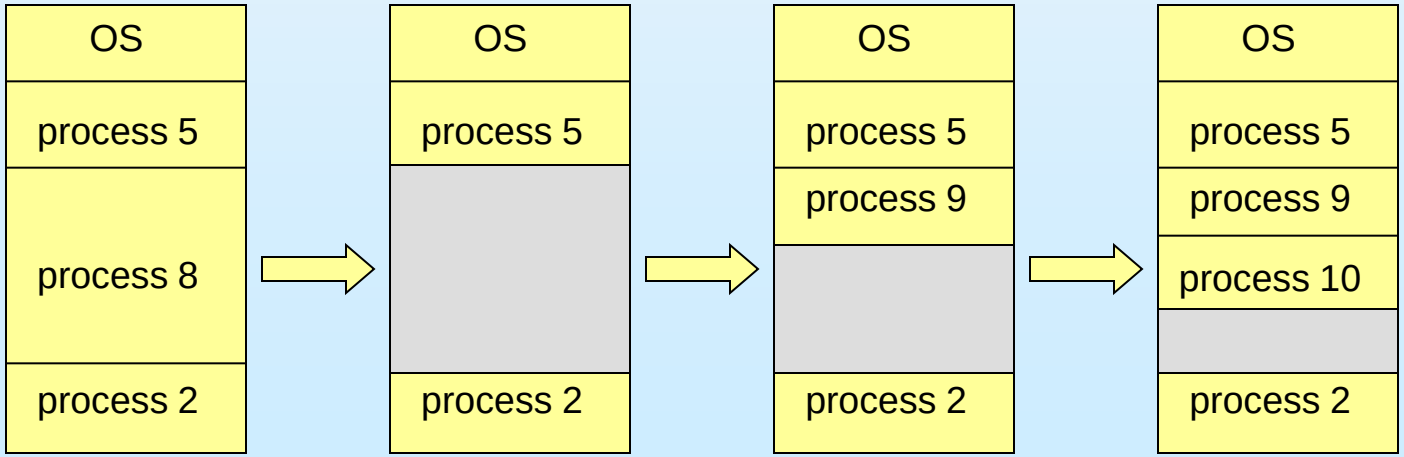
- In operating system, following are four common memory management technique
  - Contiguous Allocation : the contiguous memory allocation assigns the consecutive blocks of memory to a process requesting for memory.
    - Fixed-partitioned allocation / Static/ MFT
    - Variable-partition/ dynamic/ MVT
  - Non-Contiguous Allocation : the noncontiguous memory allocation assigns the separate memory blocks at the different location in memory space in a nonconsecutive manner to a process requesting for memory.
    - Paged memory management
    - Segmented memory management

Memory Allocation

- 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)
- Memory Allocation using MFT
  - Infixed scheme, the OS will be divided into fixed sized blocks. It takes place at the time of installation. Degree of multiprogramming is not flexible. This is because the number of blocks is fixed resulting in memory wastage due to fragmentation.

Memory Allocation using MVT

- The requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed.
- In variable partitioning scheme there are no partitions at the beginning. There is only the OS area and the rest of the available RAM. The memory is allocated to the processes as they enter. This method is more flexible as there is no internal fragmentation and there is no size limitation.



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

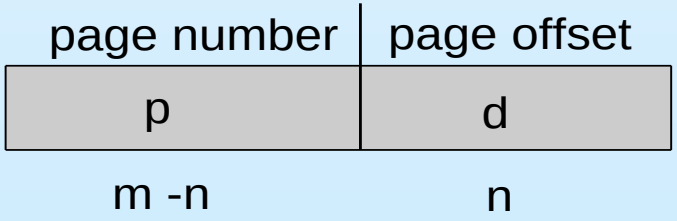
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation 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
  - I/O overhead

Paging

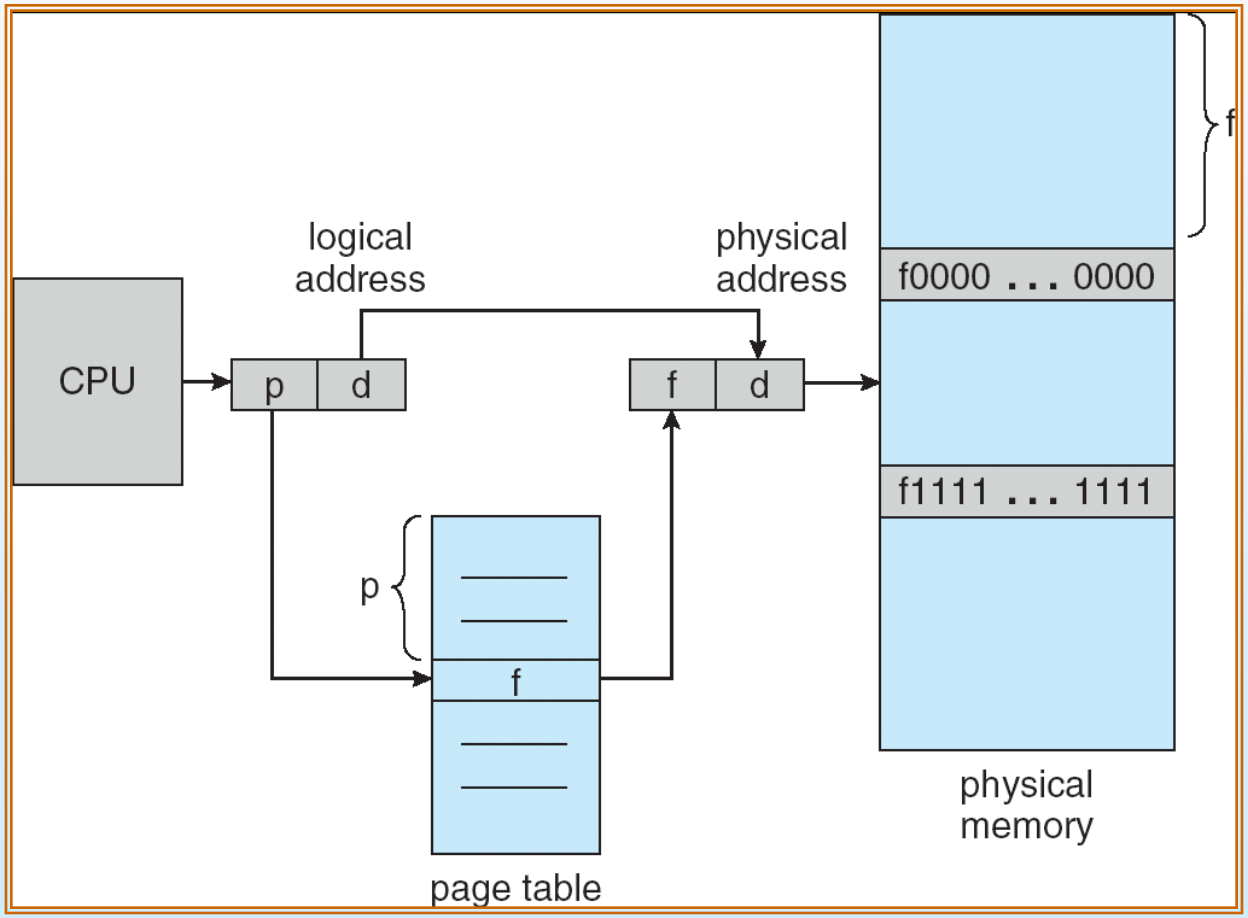
- Logical address space of a process can be **non-contiguous**; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size 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
- **Internal fragmentation**

Address Translation Scheme

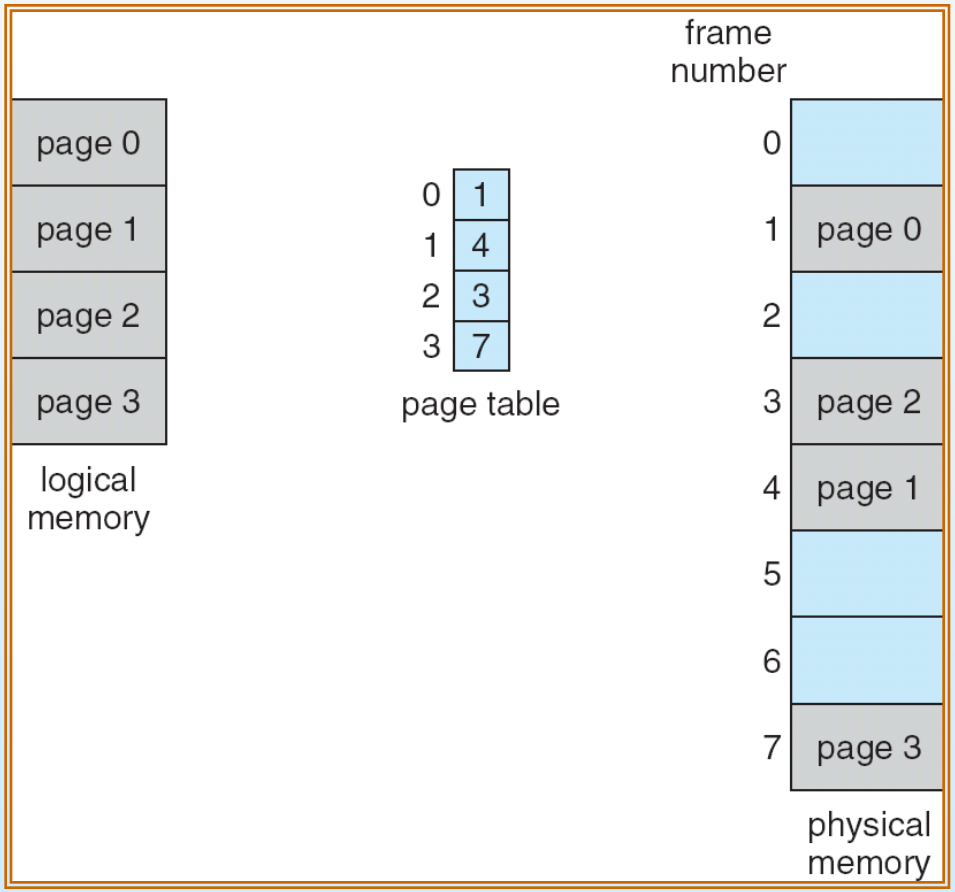
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit
  - For given logical address space 2^m and page size 2^n



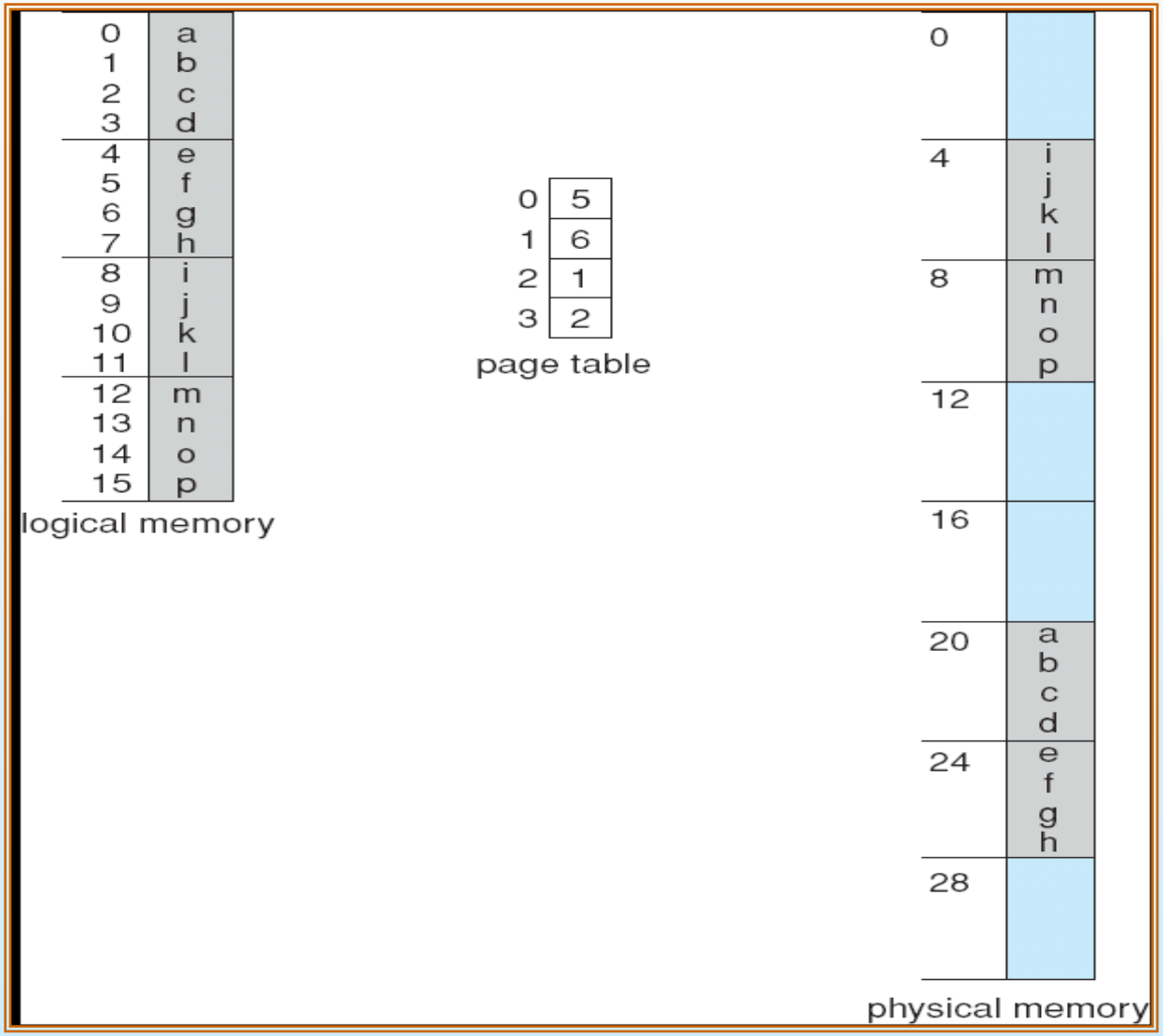
Address Translation Architecture



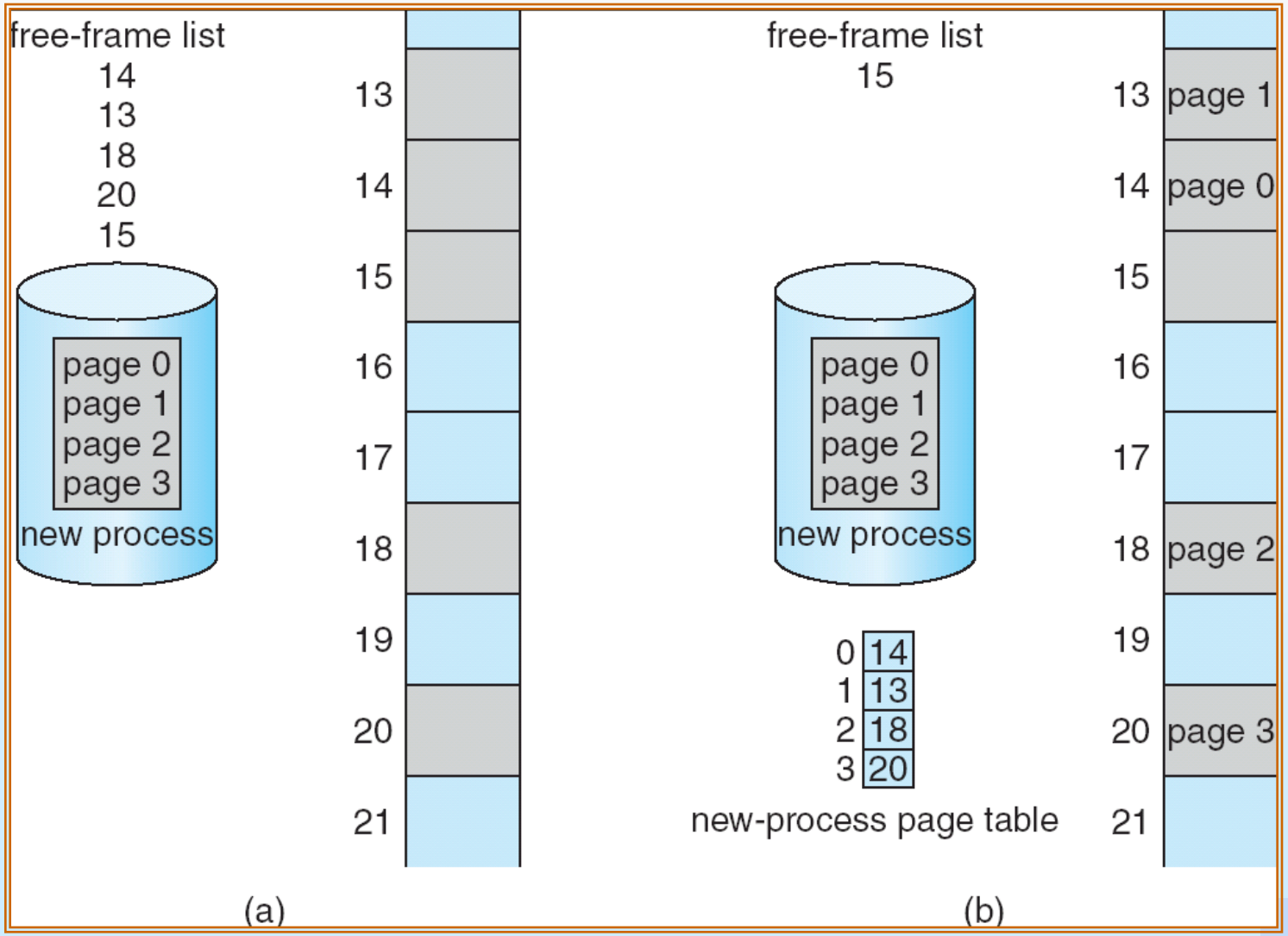
Paging Example



Paging Example



Free Frames

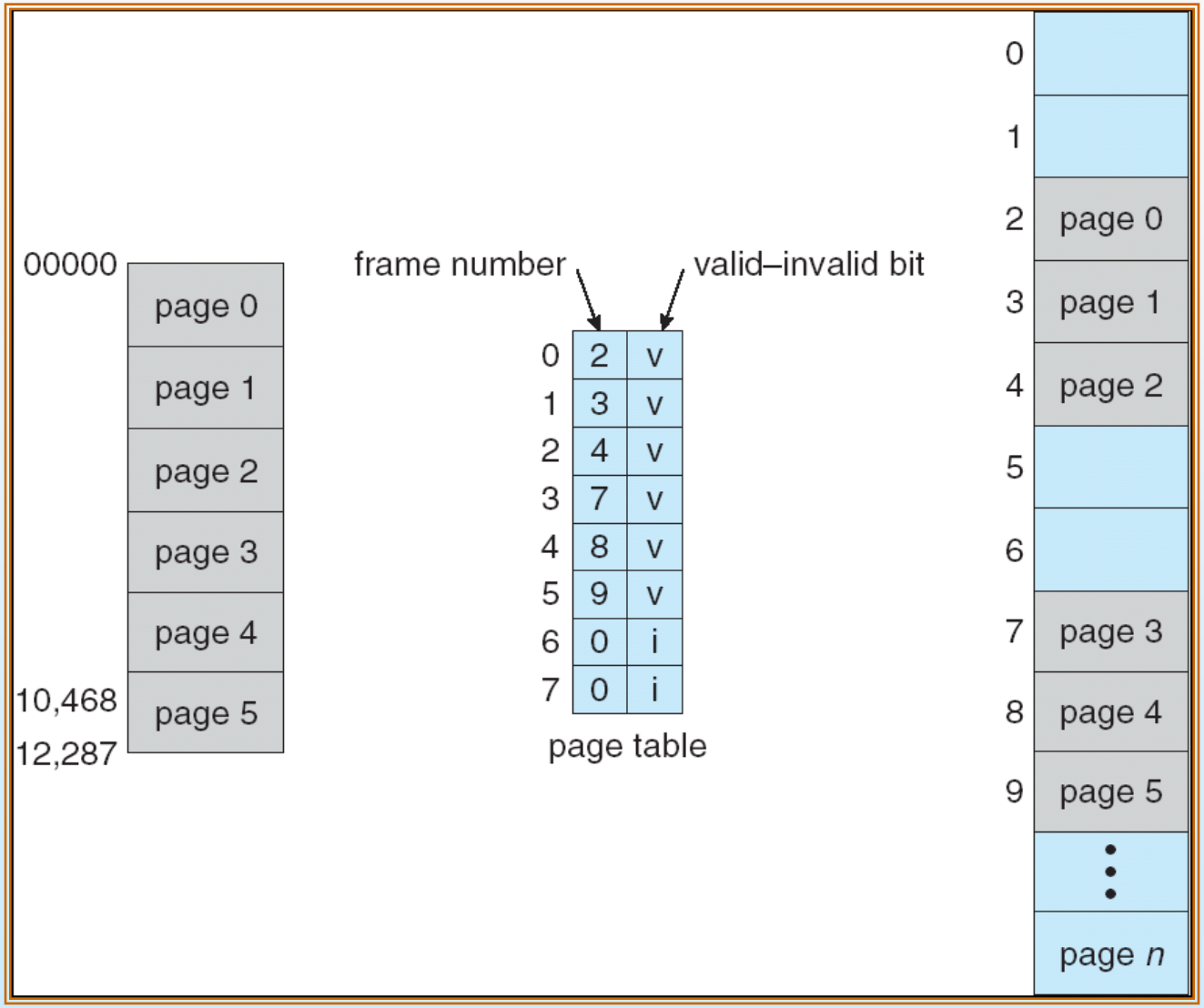

free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Memory Protection

- 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

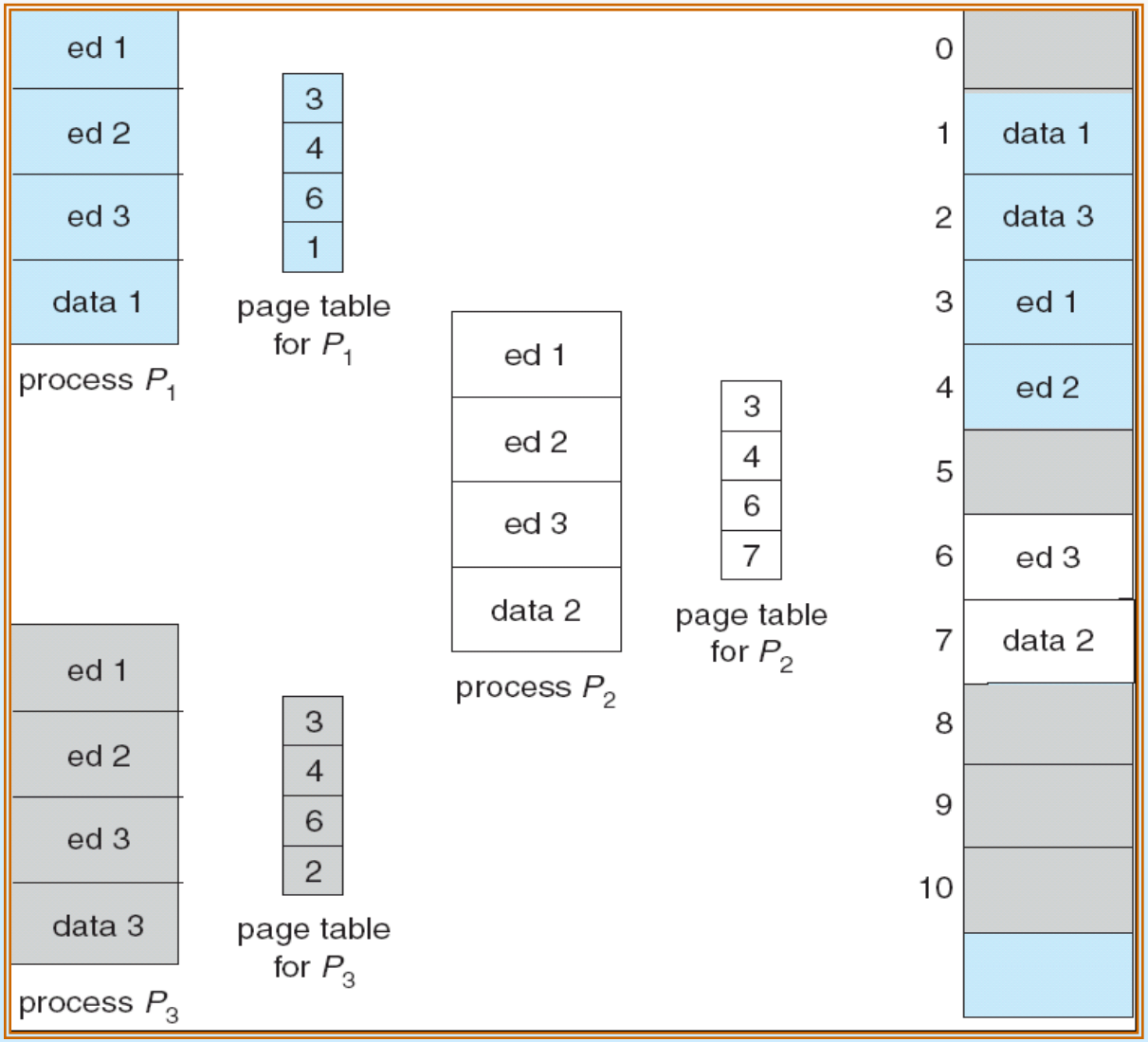


Shared Pages

- **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



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,

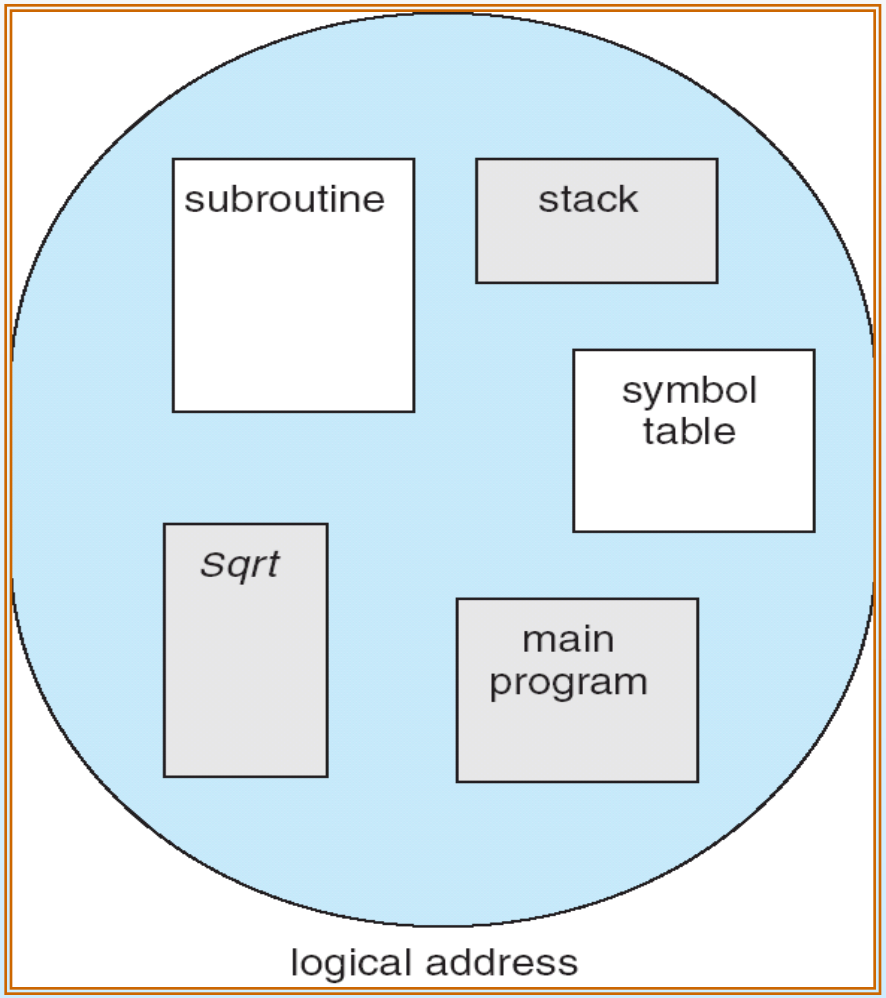
  local variables, global variables,

  common block,

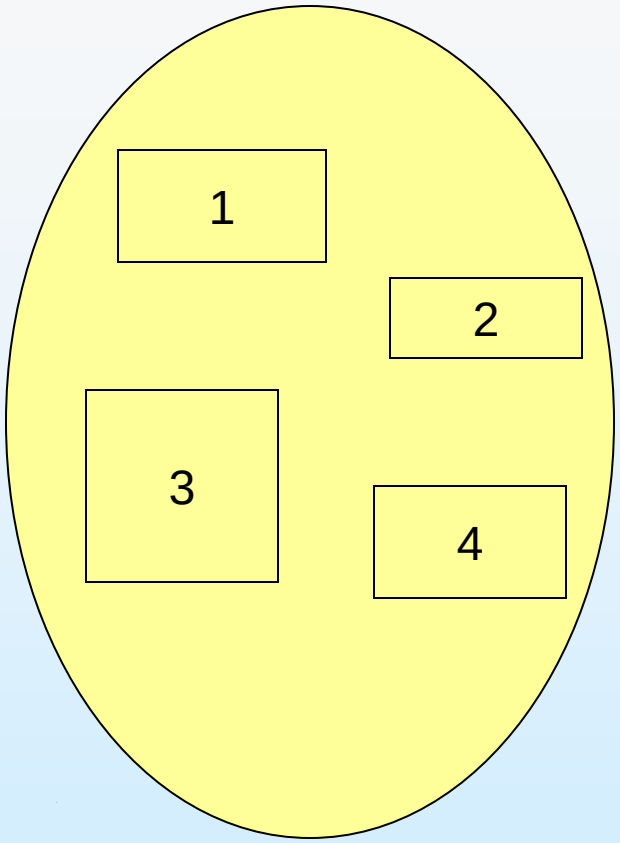
  stack,

  symbol table, arrays

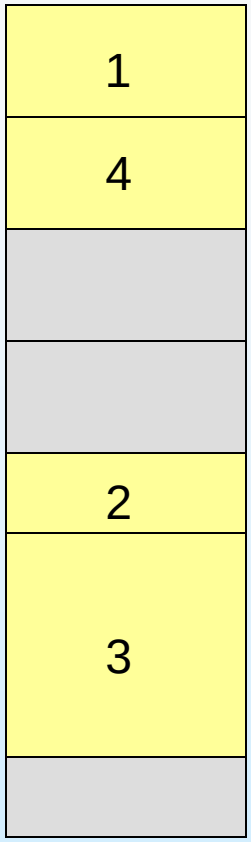
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

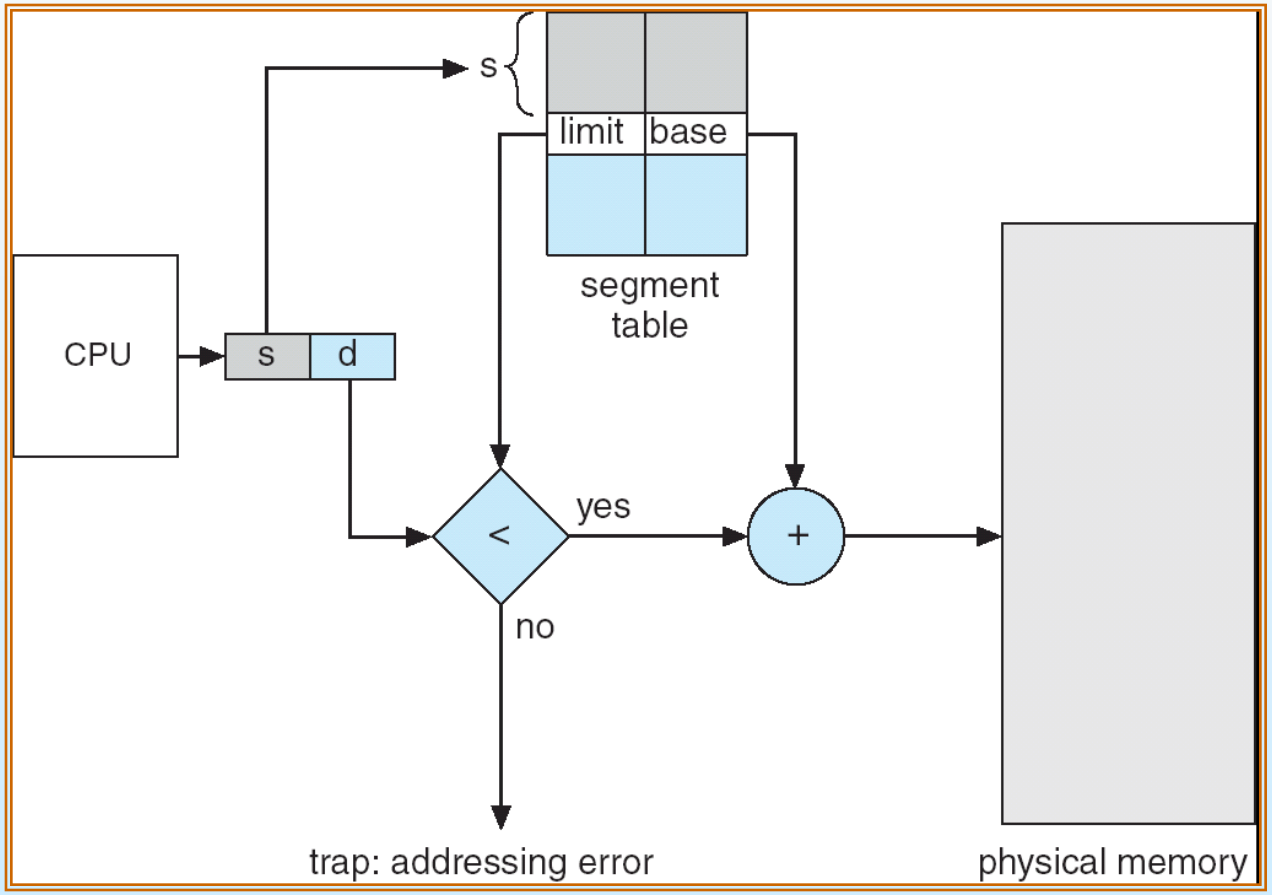
- Logical address consists of a two tuple:

  <segment-number, offset>,

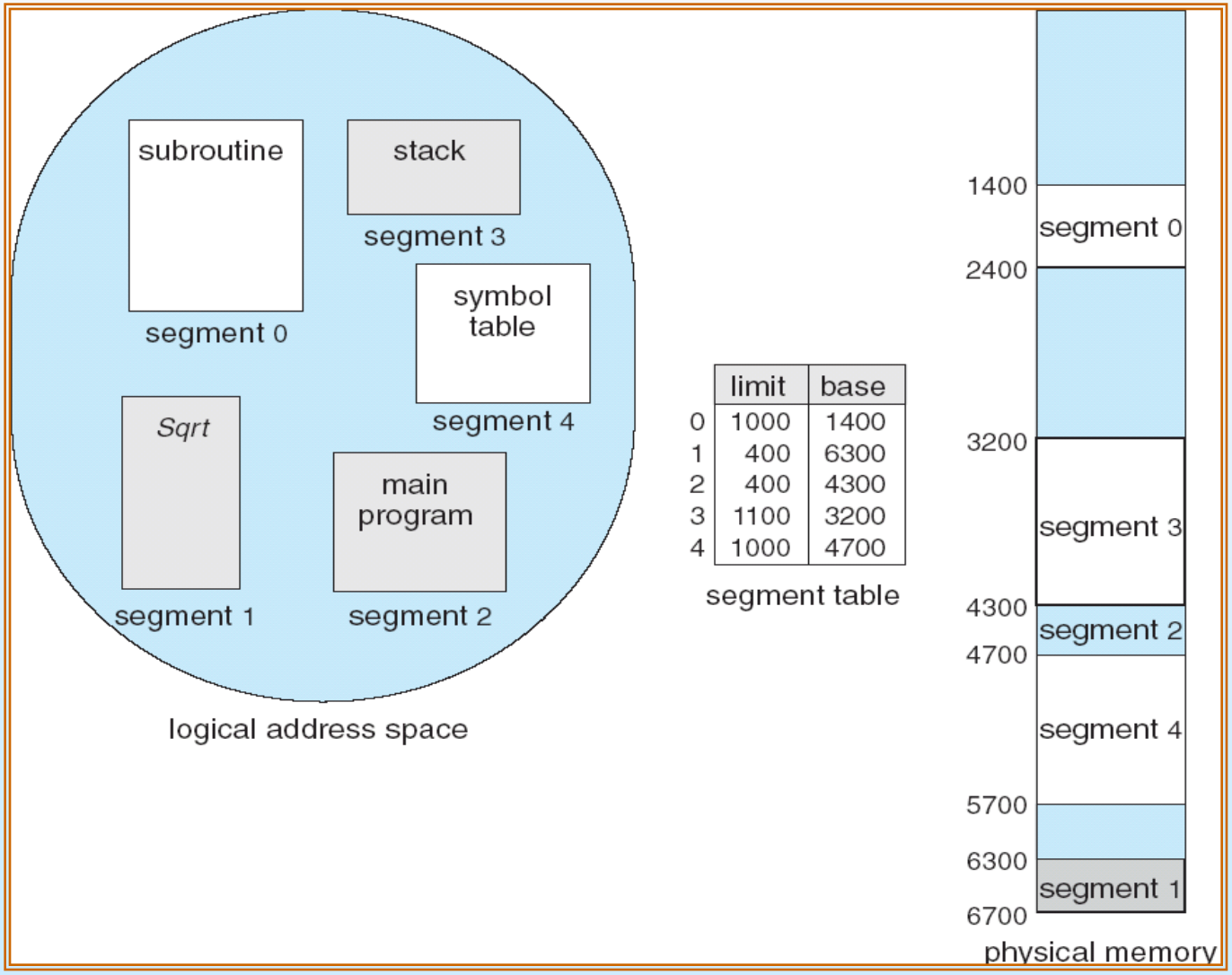
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - *limit* – specifies the length of the segment
- *Segment-table base register (STBR)* points to the segment table's location in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

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

Address Translation Architecture



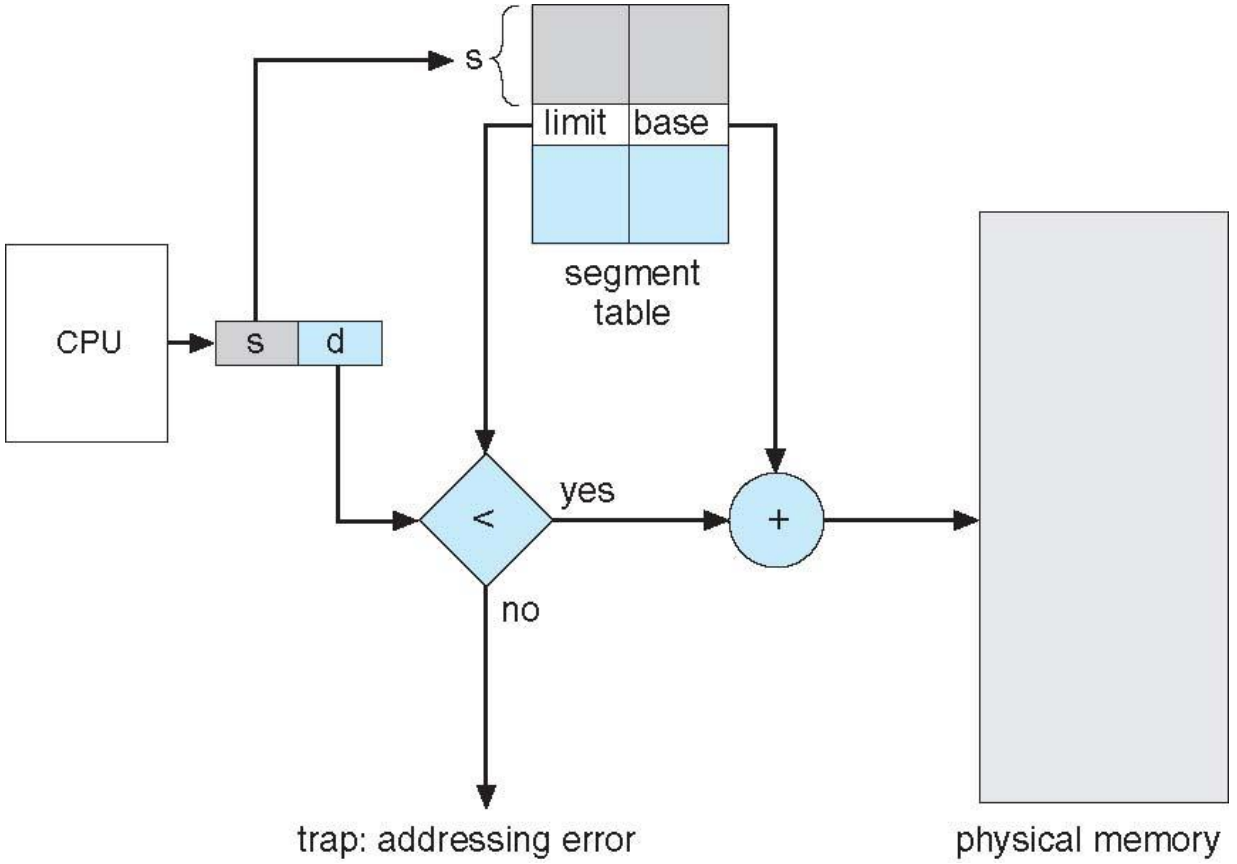
Example of Segmentation

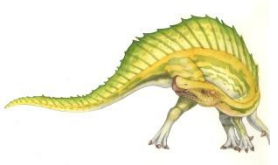


Segmentation Architecture (Cont.)

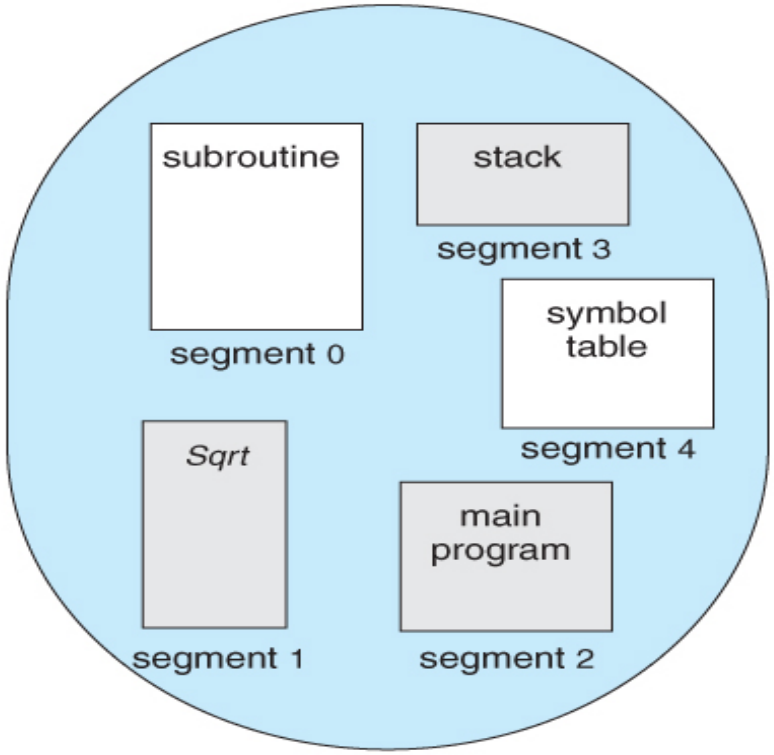
- Protection. With each entry in segment table associate:
  - validation bit = 0 $\Rightarrow$ illegal segment
  - read/write/execute privileges
- Since segments vary in length, memory allocation is a **dynamic storage-allocation problem**
- Segmentation leads to **external fragmentation**
- Code sharing occurs at segment level; shared segments

Segmentation Hardware





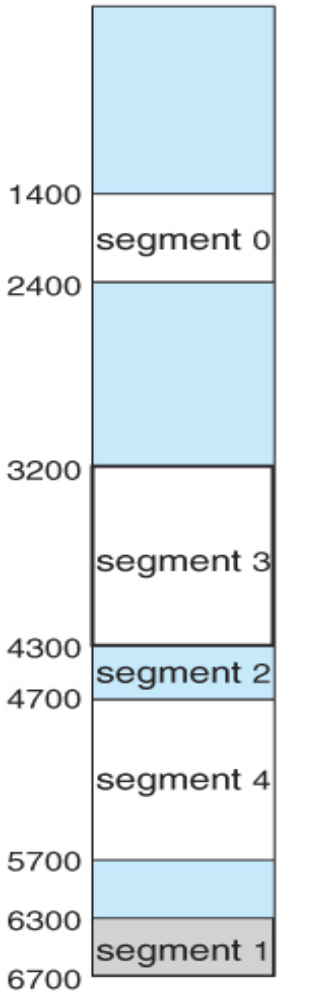
Example of segmentation



logical address space

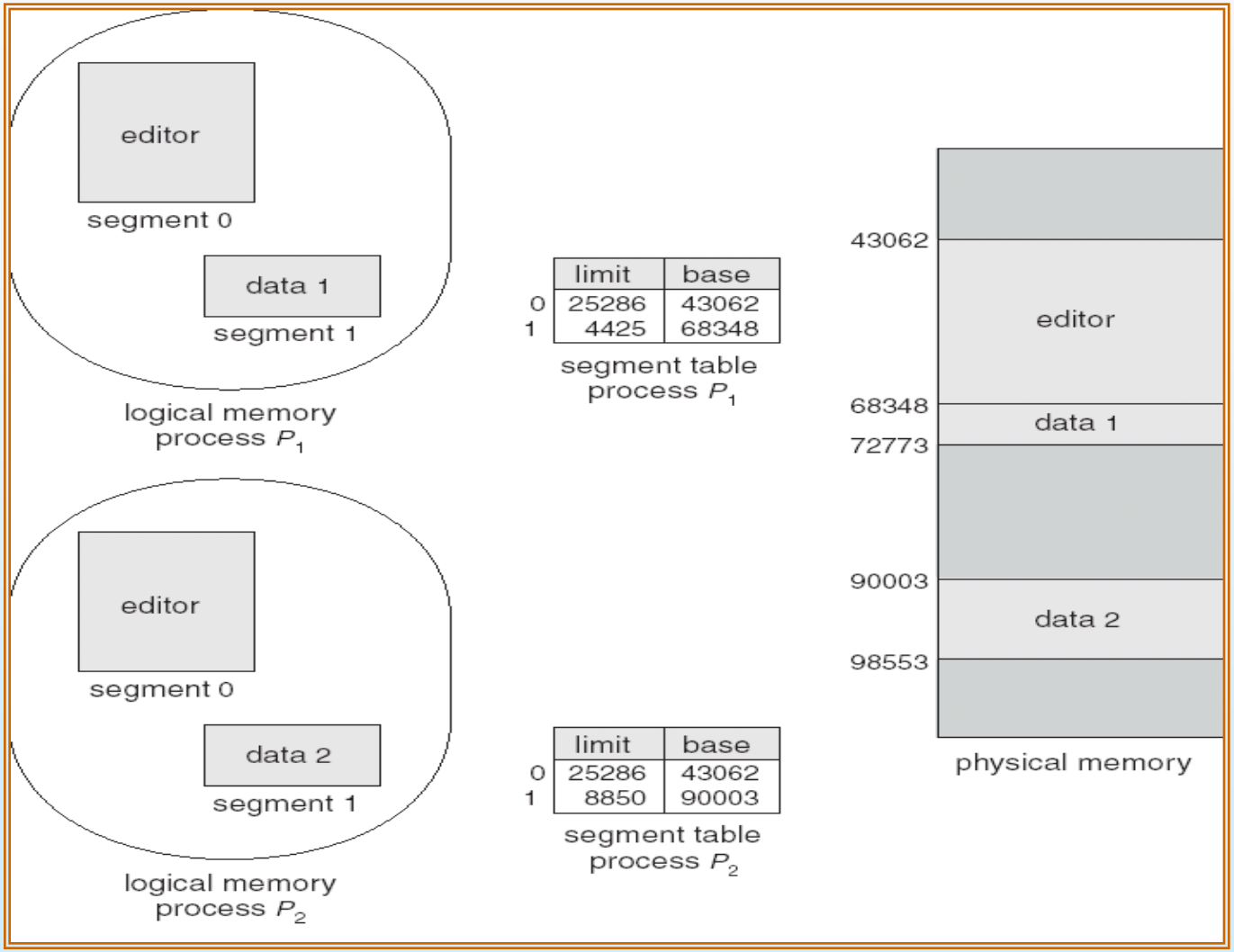
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



physical memory

Sharing of Segments



End of Chapter 8