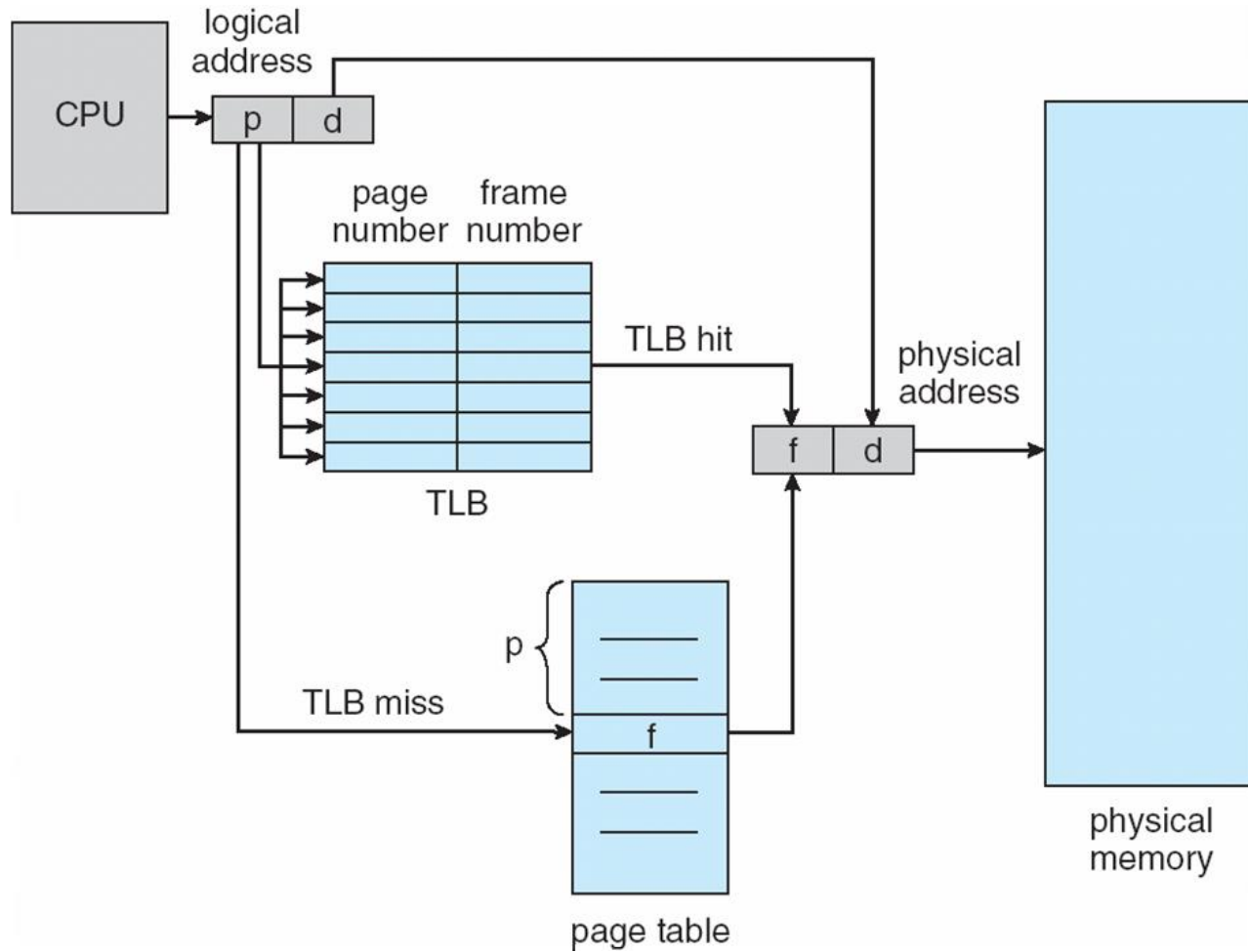# Main Memory III

Paging Hardware With TLB

TLB Hardware is shared by multiple processes

- Problem: when a process is swapped out and a new process is swapped in, the content of the TLB becomes outdated

- Solutions:
    - flush the TLB when process switch; or
    - store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Implementation of Page Table

- Page table is kept in main memory (kernel space)
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- Memory protection implemented by
  - PTLR specifies the length of page table (the number of pages for a process)
  - If the page number of an address >= the value of PTLR, the logical address is invalid

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- **Private code and data**
  - Each process keeps a separate copy of the code and data

Structure of the Page Table

- Number of logical pages could be very large →

- Hierarchical Paging
- Inverted Page Tables
- Hashed Page Tables

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- The page table should be 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:

page number		page offset
p_i	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

7

Address-Translation Scheme

logical address

p_1	p_2	d

p_1 { outer page table

p_2 { page of page table

d {
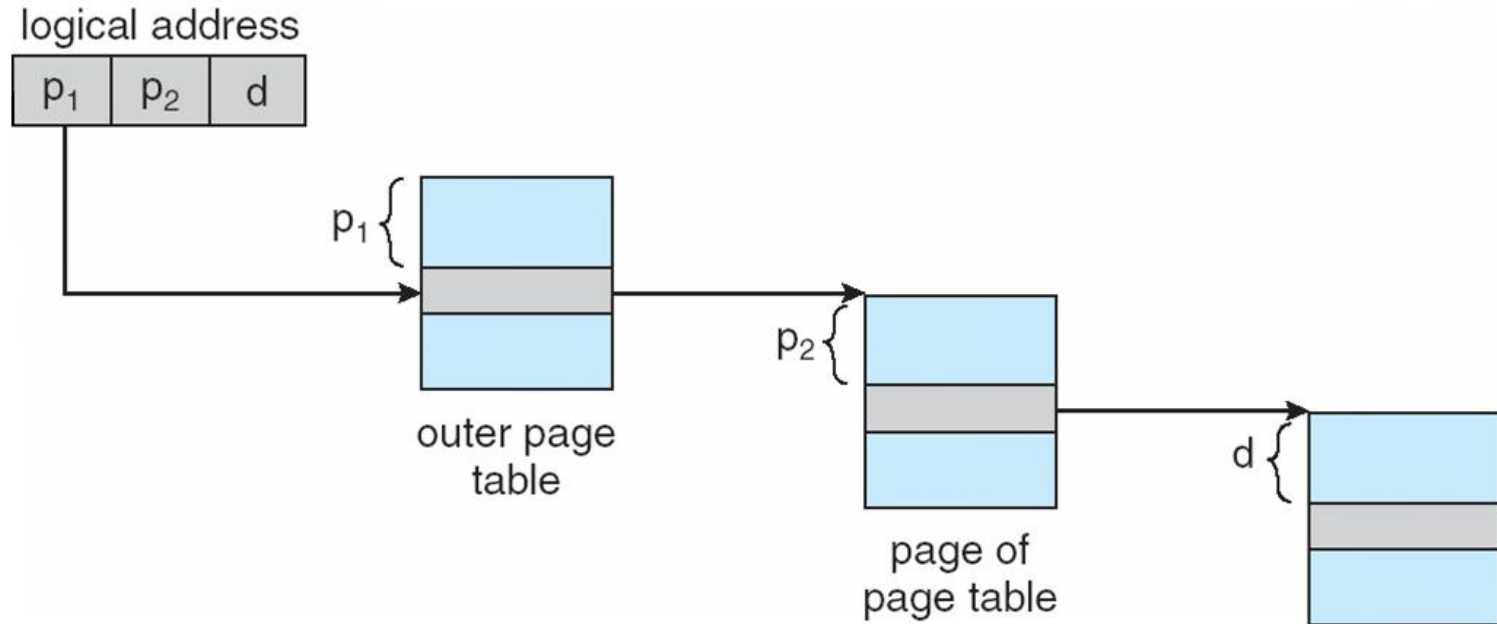
Example: 2-layer Page Table

Process Logical address space

Main Memory

	Page
Page 0	
Page 1	
Page 2	
Page 3	
Page 4	
Page 5	
Page 6	
Page 7	
Page 8	
Page 9	
Page 10	
Page 11	
Page 12	
Page 13	
Page 14	
Page 15	

Page size = Frame size = 8 bytes

?

	Page
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
…	
$2^{16}-1$	

Example: 2-layer Page Table

Outer Page Table

outer Page #	
0	
1	
2	
3	

Inner Page Table 0

Inner Page #	frame#
0	3
1	7
2	…
3	…

Inner Page Table 1

Inner Page #	frame#
0	…
1	11
2	…
3	0

Inner Page Table 2

Inner Page Table 3

	Page
0	7
1	
2	
3	0
4	
5	
6	
7	1
8	
9	
10	
11	5
12	
13	
…	
$2^{16}-1$	

Main Memory

Page size = 8 bytes

Logical address:
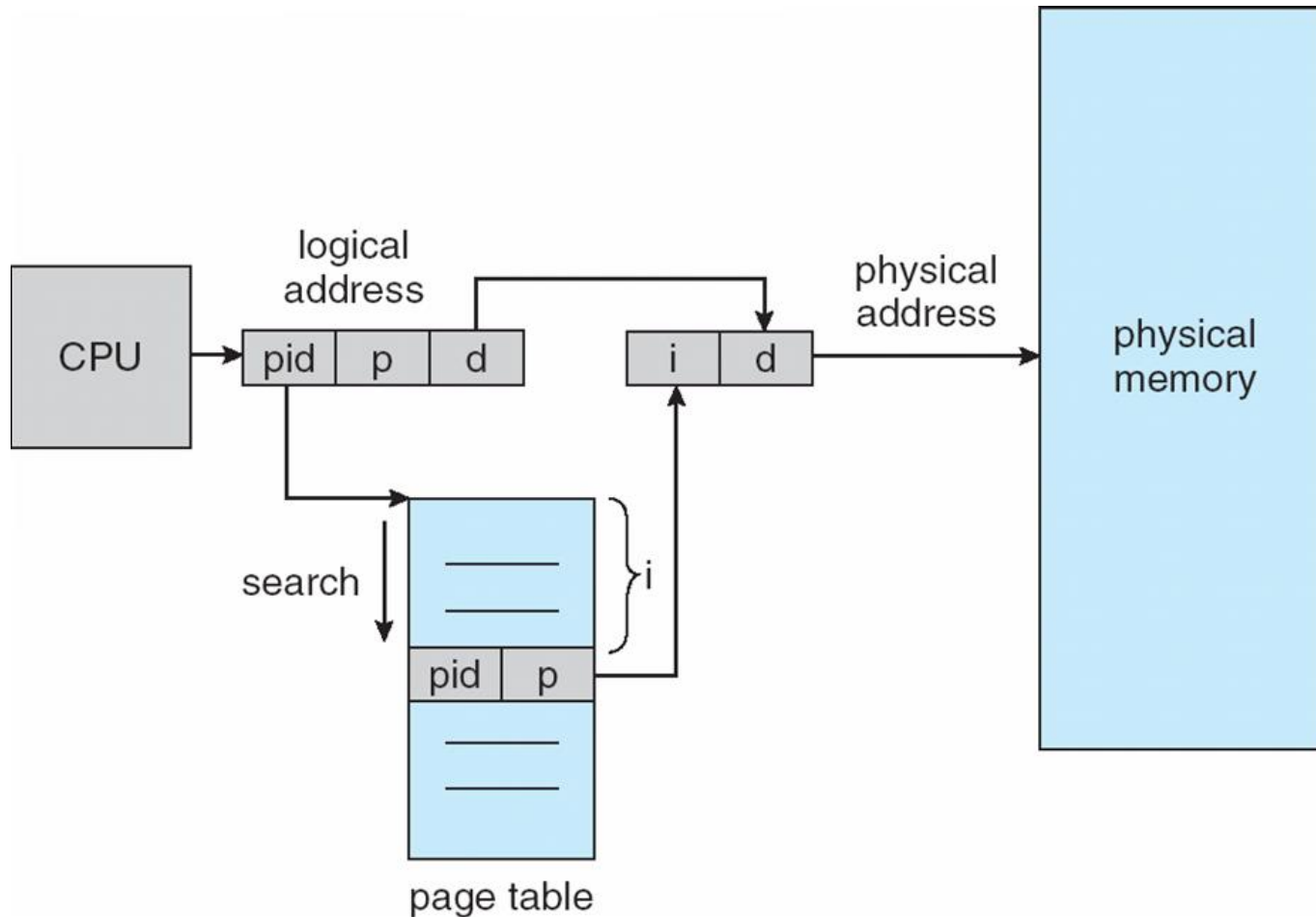0001001

00: outer-page #
01: inner-page #
001: page offset

frame# = $(7)_{10}$ = $(111)_2$

Physical address:
111001

Inverted Page Table

- One entry for each **frame** of physical 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)

Inverted Page Table Architecture

Process 1's Logical address space

| Page 0 |
| Page 1 |
| Page 2 |
| Page 3 |
| Page 4 |

Proc 1's Page Table

	frame #
0	7
1	5
2	10
3	12
4	0

Main Memory

	Page
0	P1:Page4
1	P2:Page0
2	P3:Page 0
3	P2:Page1
4	P4:Page1
5	P1:Page1
6	P4:Page0
7	P1:Page0
8	P2:Page3
9	P3:Page1
10	P1:Page2
11	P5:Page1
12	P1:Page3
13	P5:Page0
14	P5:Page2
15	P2:Page2

Process 2's Logical address space

| Page 0 |
| Page 1 |
| Page 2 |
| Page 3 |

Proc 2's Page Table

	frame #
0	1
1	3
2	15
3	8

Process 1's Logical address space

Page 0
Page 1
Page 2
Page 3
Page 4

Process 2's Logical address space

Page 0
Page 1
Page 2
Page 3

Inverted Page Table

Frame #	Proc #	Page #
0	1	4
1	2	0
2	3	0
3	2	1
4	4	1
5	1	1
6	4	0
7	1	0
8	2	3
9	3	1
10	1	2
11	5	1
12	1	3
13	5	0
14	5	2
15	2	2

Main Memory

	Page
0	P1:Page4
1	P2:Page0
2	P3:Page 0
3	P2:Page1
4	P4:Page1
5	P1:Page1
6	P4:Page0
7	P1:Page0
8	P2:Page3
9	P3:Page1
10	P1:Page2
11	P5:Page1
12	P1:Page3
13	P5:Page0
14	P5:Page2
15	P2:Page2

Hashed Inverted Page Tables

- The virtual page number is hashed into a table
  - Each entry of this 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
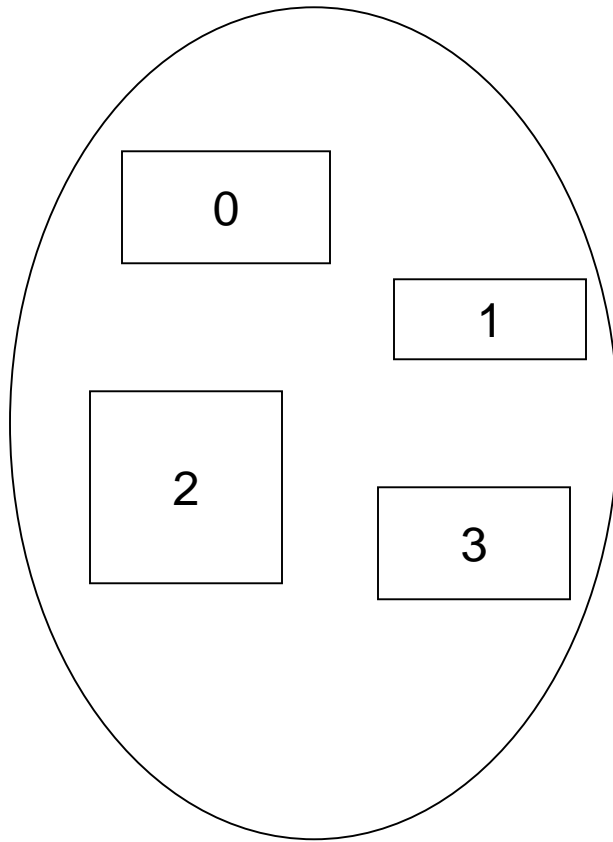
Hashed Page Table
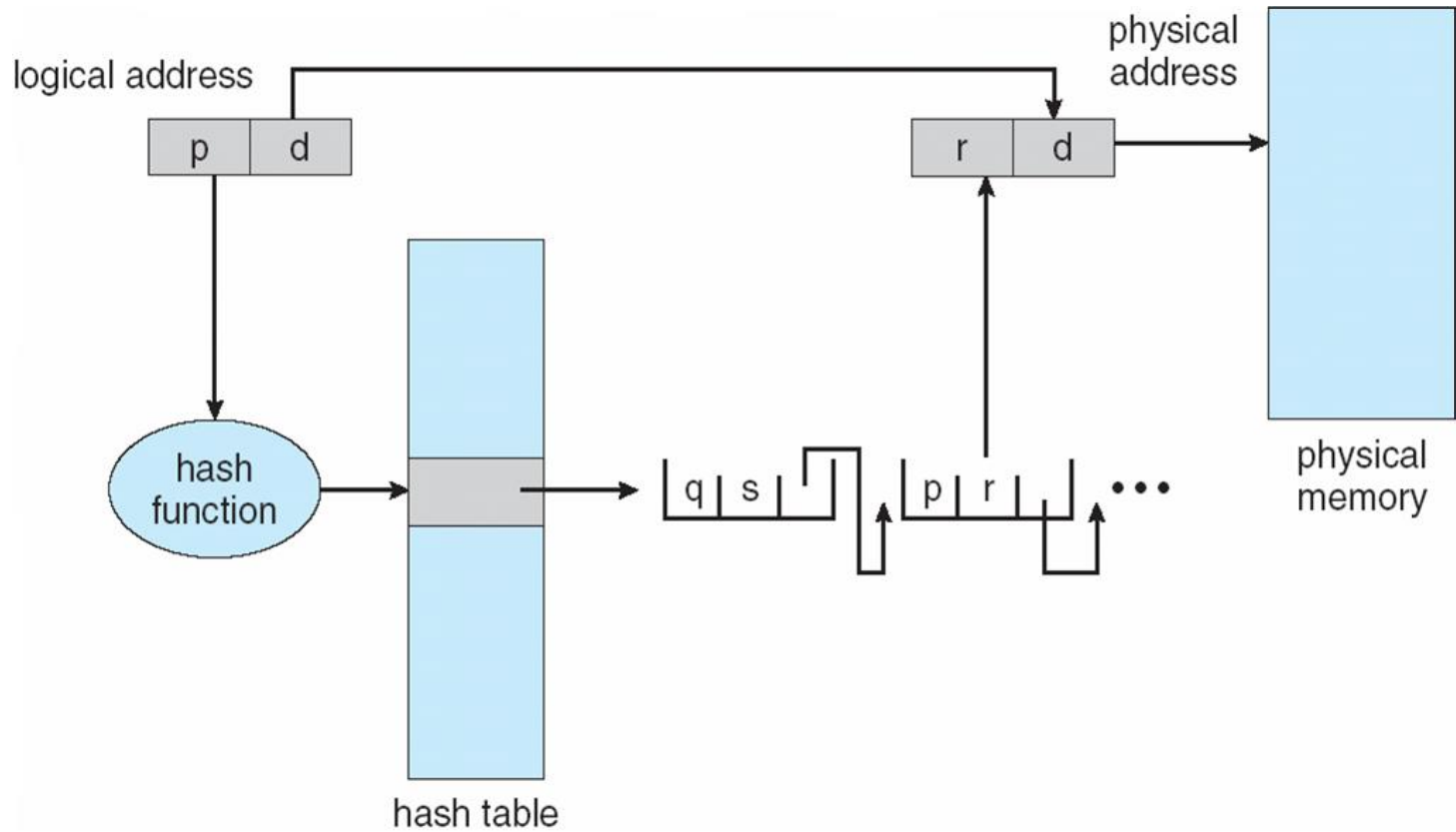
Segmentation

- 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,

        common block,

        stack (local variables),

        heap (global variables),

        …
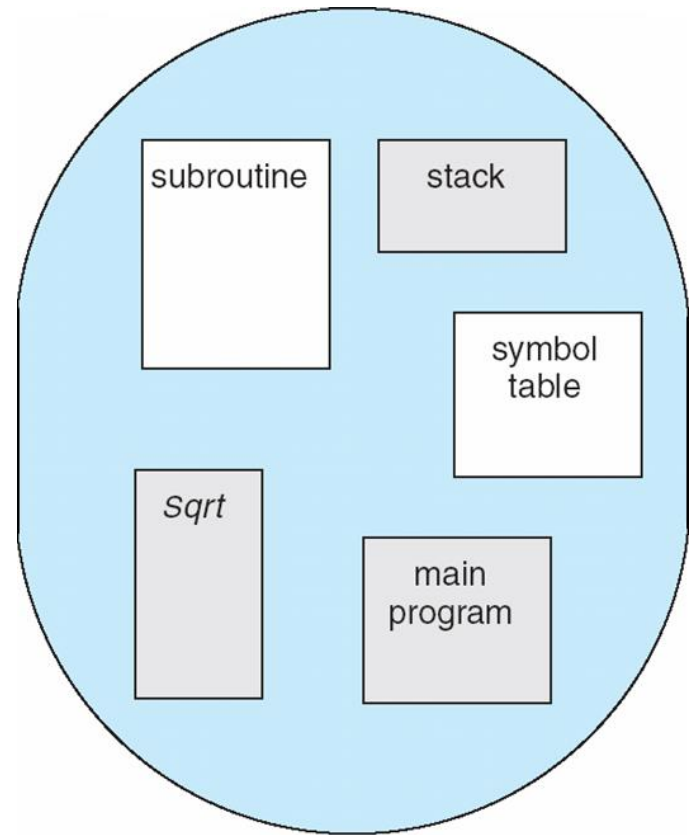
logical address

Logical View of Segmentation
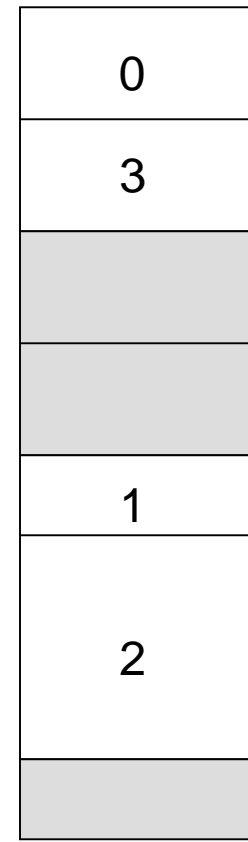
user space   physical memory space

Segmentation Architecture

- **Segment table** – maps segments to memory; each 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
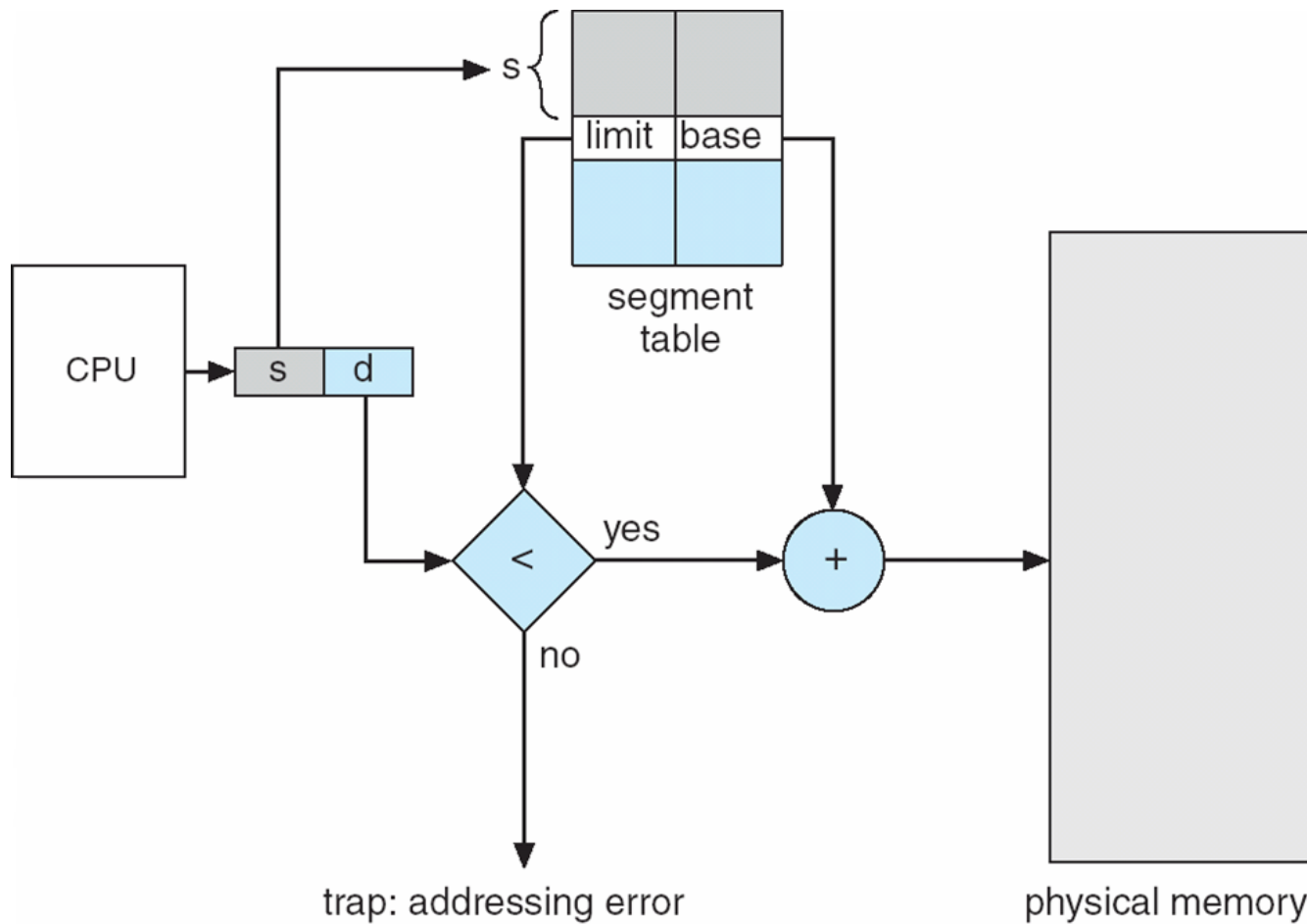- **Segment-table length register (STLR)** indicates number of segments used by a program

Example of Segmentation
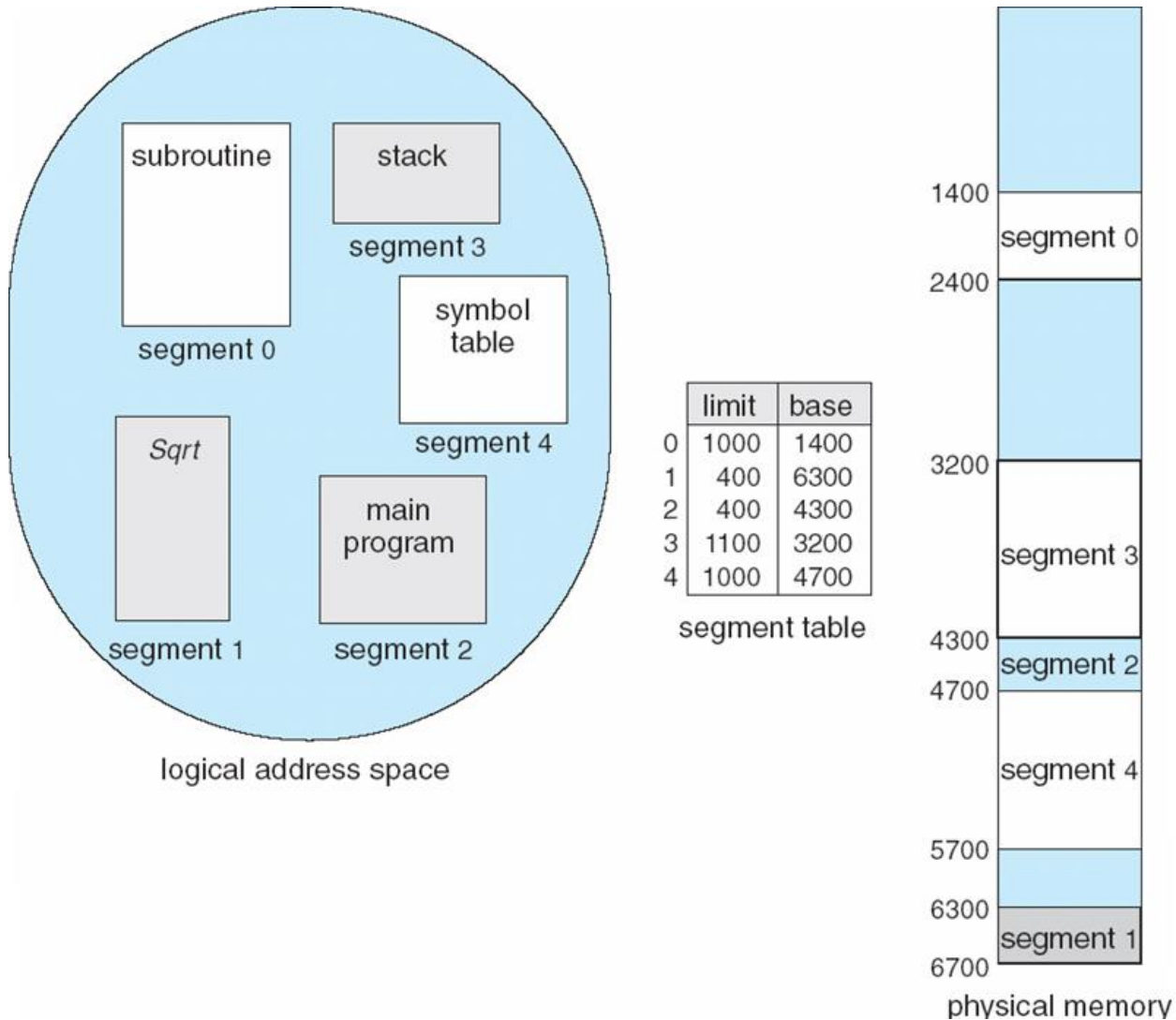
Segmentation Architecture

- Logical address consists of a two tuple: <segment-number, offset>
- Segment number s is legal if s < **STLR**

Segmentation Hardware

Segmentation Architecture

- Protection
    - With each entry in segment table associate:
        - read/write/execute privileges
        - validation bit $= 0 \Rightarrow$ illegal segment
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
    - Could have external fragmentation problem: To address the problem, combination of segmentation and paging may be applied