

# Memory Management

Lectures notes from the text  
supplement by Siberschatz and Galvin  
Modified by B.Ramamurthy  
Chapter 8

# Background

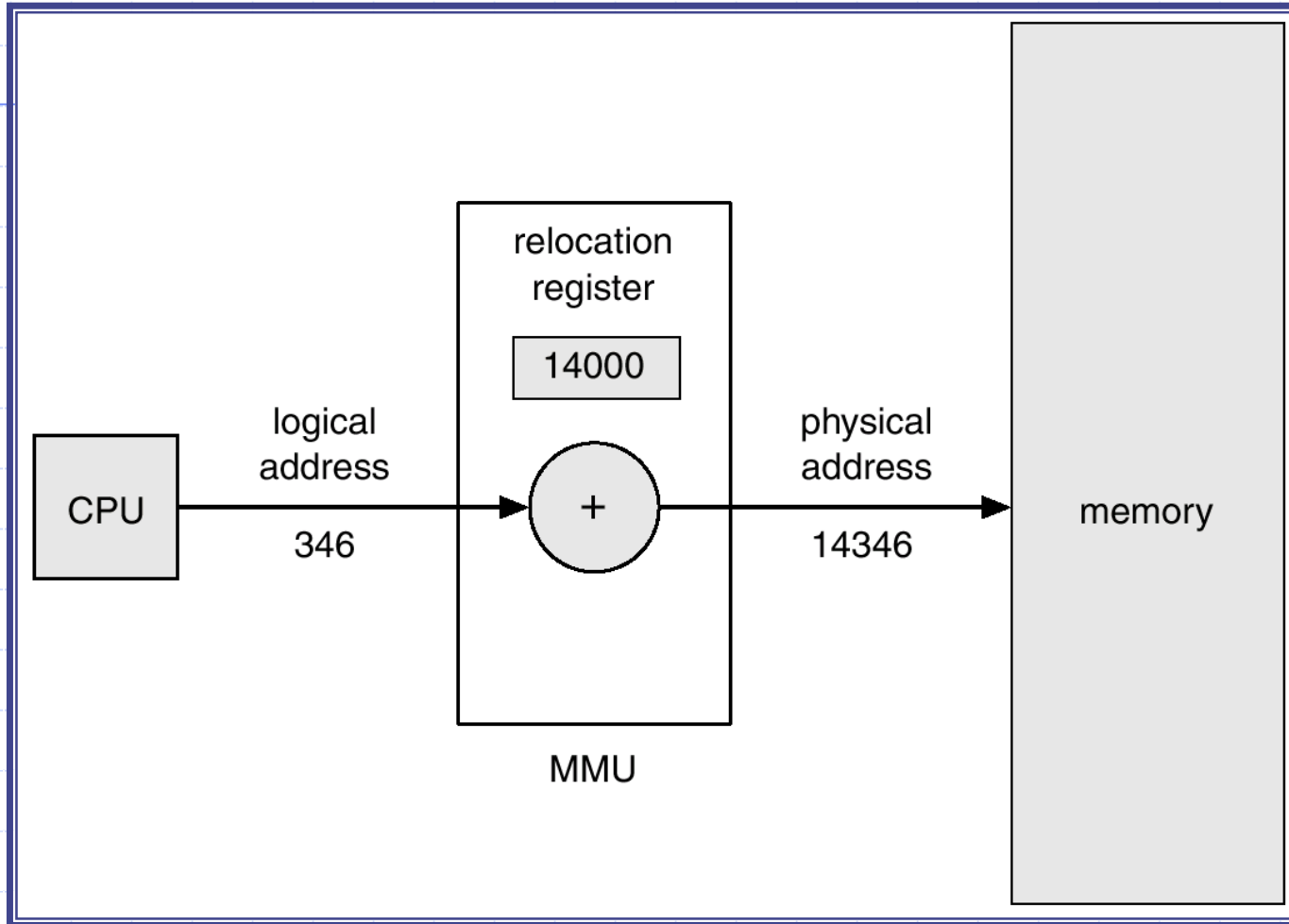
- ◆ Program must be brought into memory and placed within a process for it to be run.
- ◆ *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- ◆ User programs go through several steps before being run.

# 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*).

# Dynamic relocation using a relocation register



# 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.

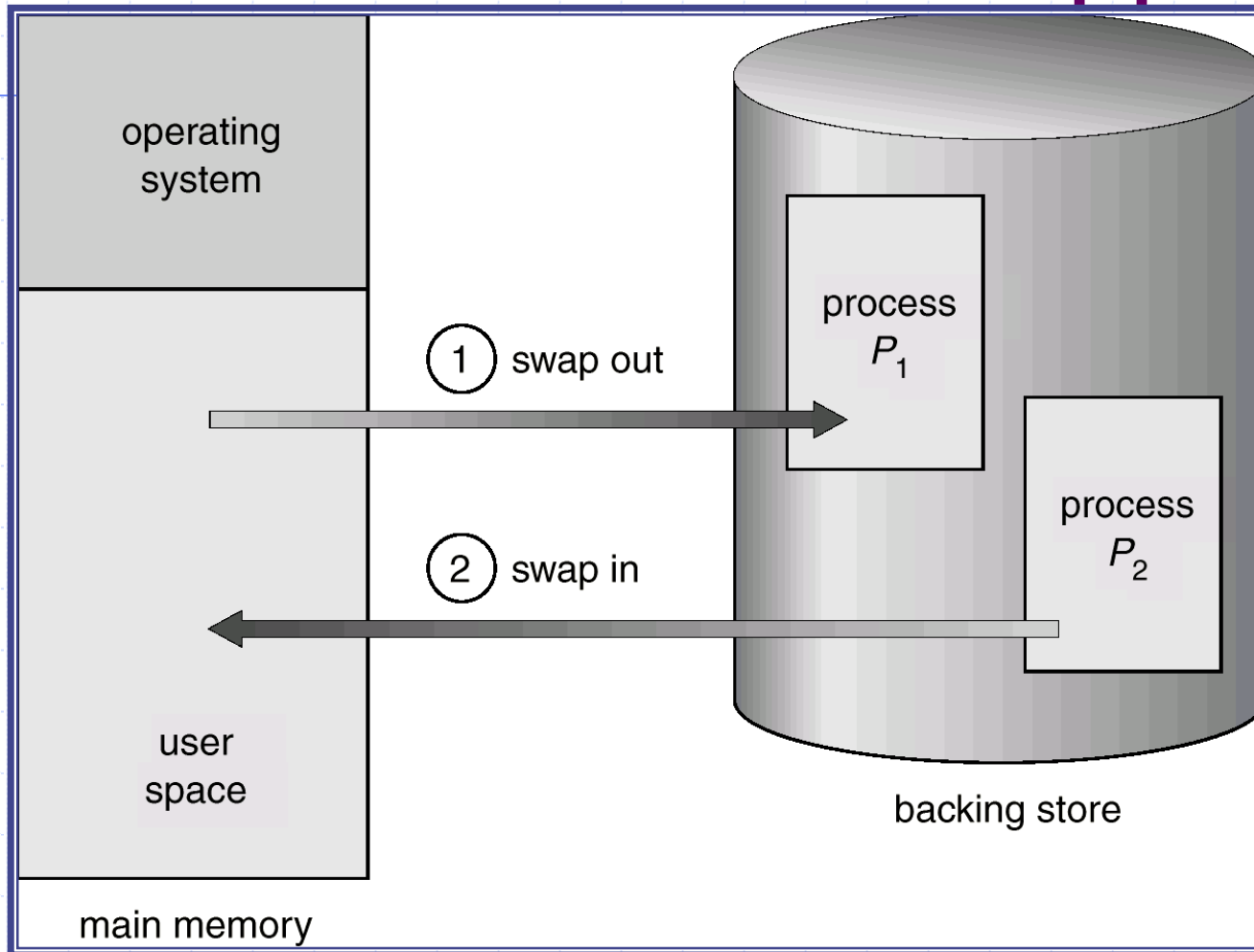
# Overlays

- ◆ Keep in memory only those instructions and data that are needed at any given time.
- ◆ Needed when process is larger than amount of memory allocated to it.
- ◆ Implemented by user, no special support needed from operating system, programming design of overlay structure is complex

# 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.

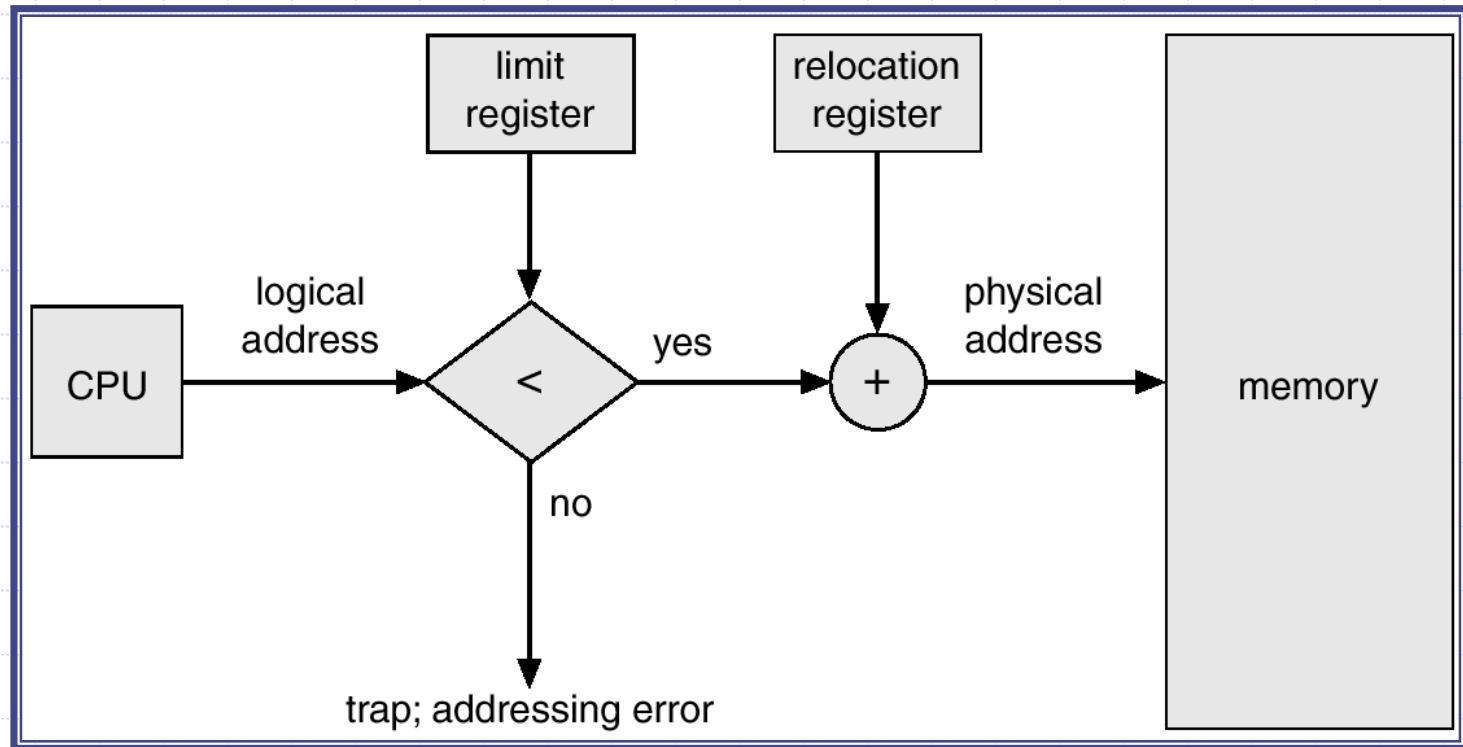
# Schematic View of Swapping



# 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.
- ◆ Single-partition allocation
  - Relocation-register scheme 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.

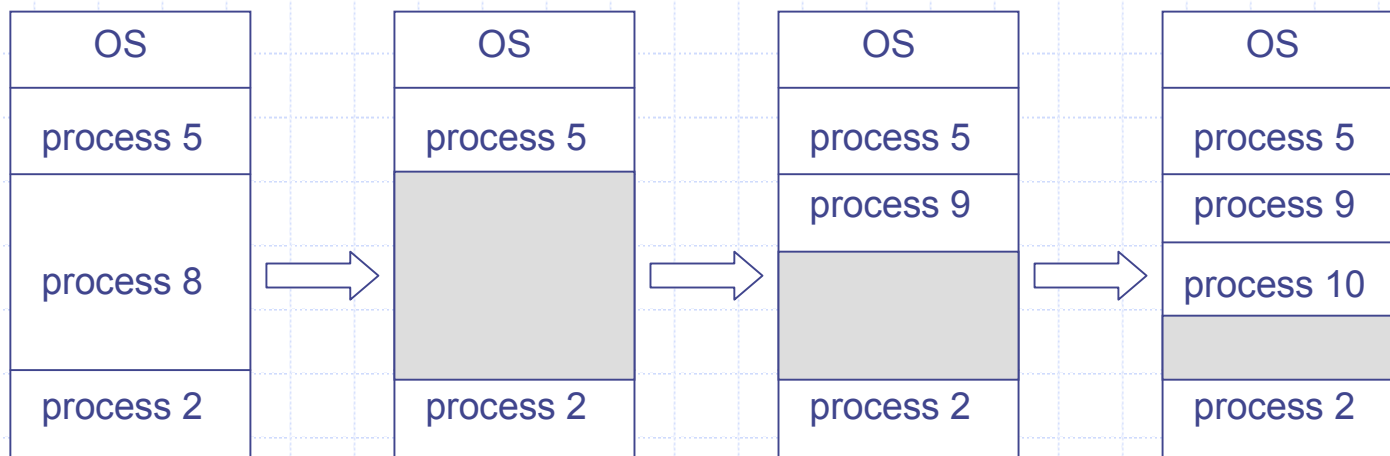
# Hardware Support for Relocation and Limit Registers



# 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

- ◆ **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 problem
    - ◆ Latch job in memory while it is involved in I/O.
    - ◆ Do I/O only into OS buffers.

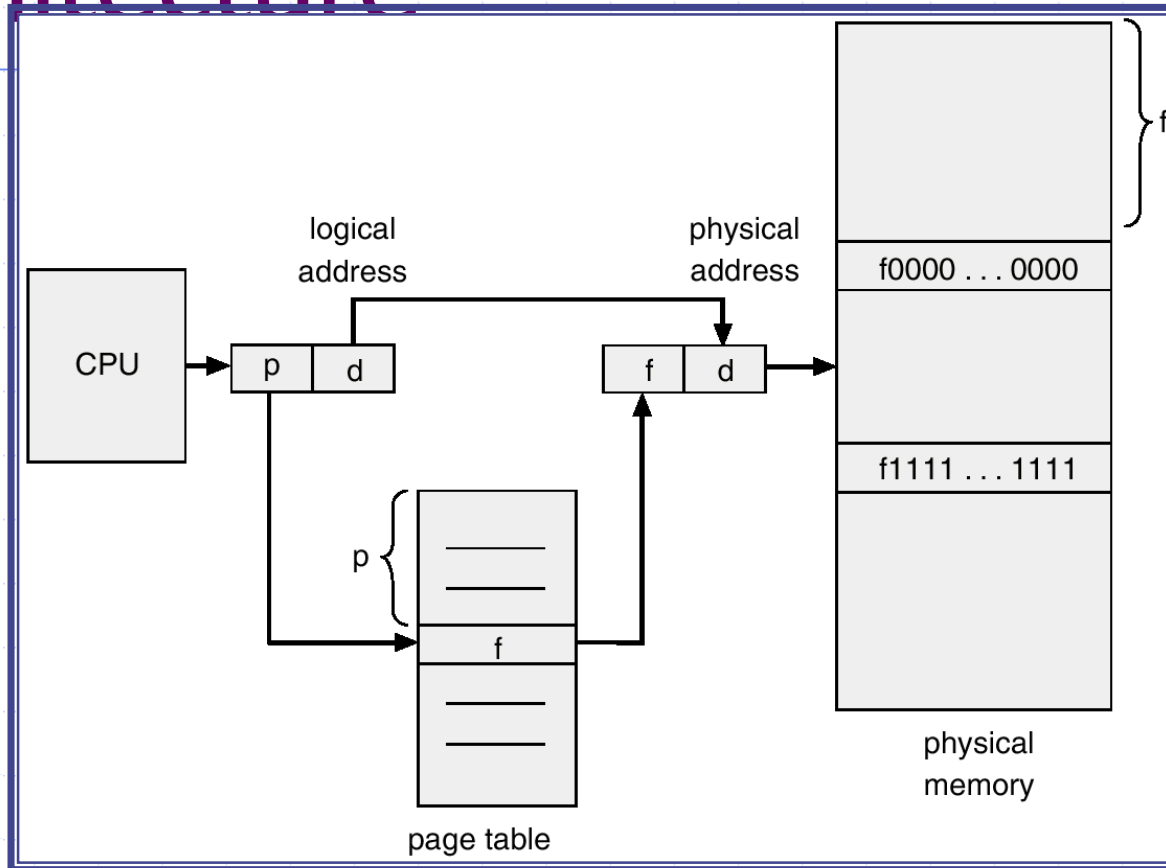
# Paging

- ◆ 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 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.

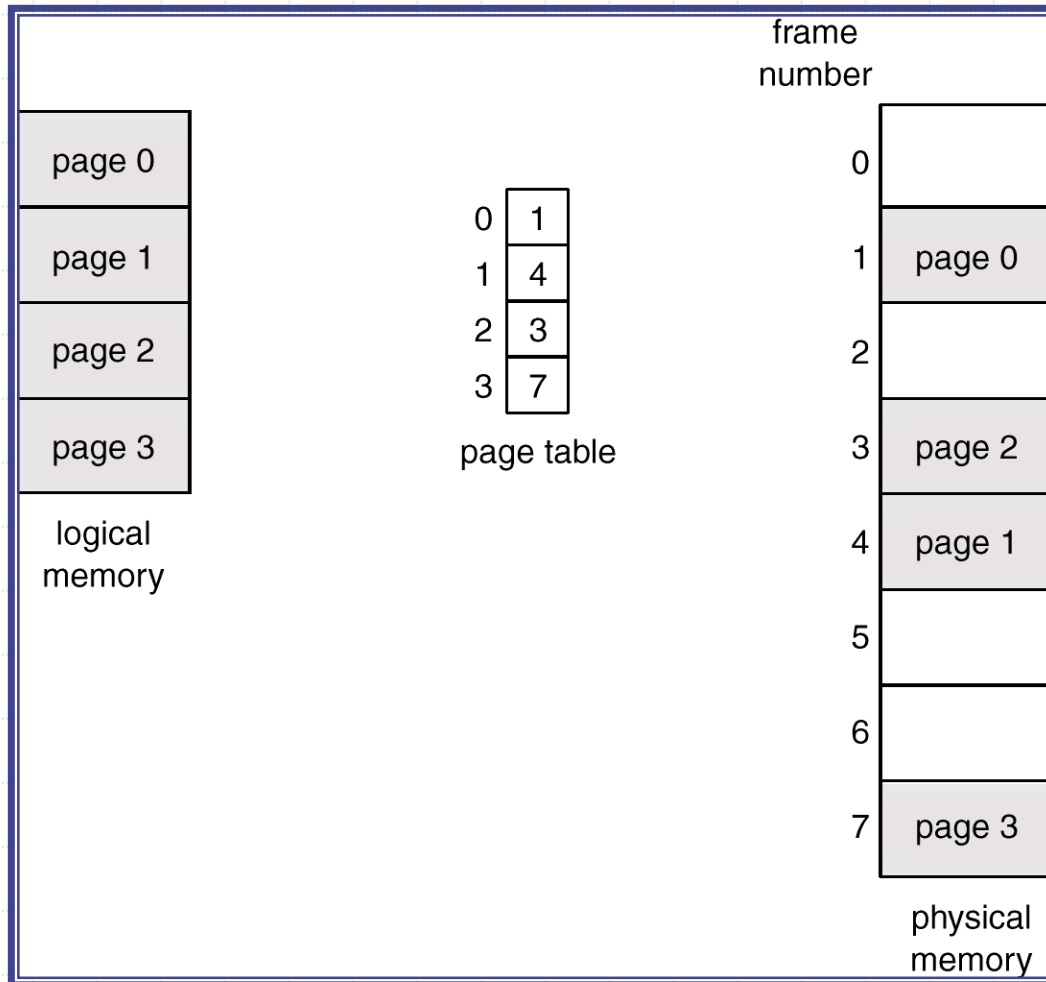
# Address Translation Scheme

- ◆ 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.

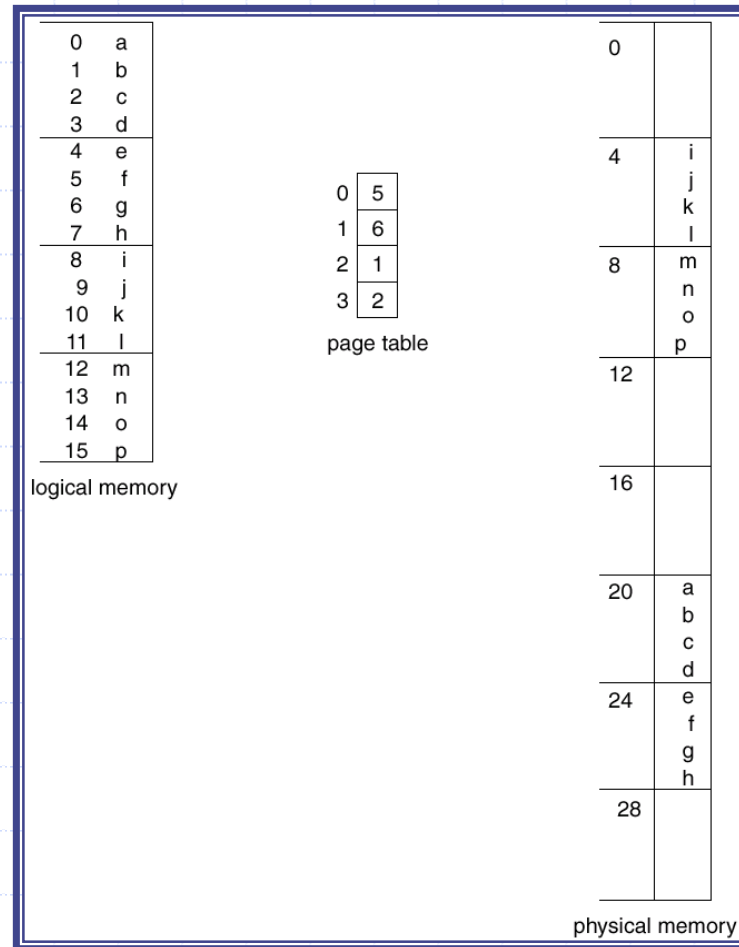
# Address Translation Architecture



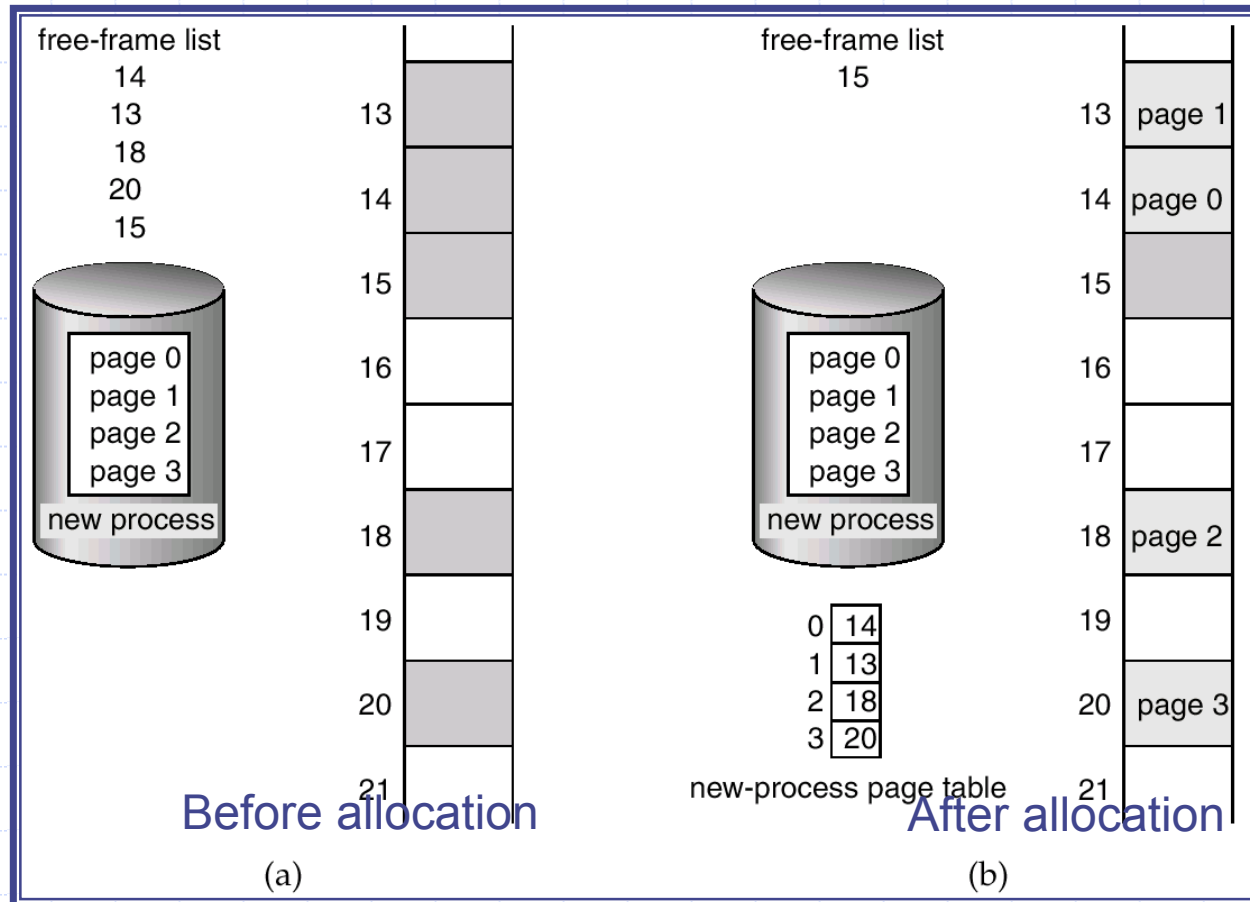
# Paging Example



# Paging Example



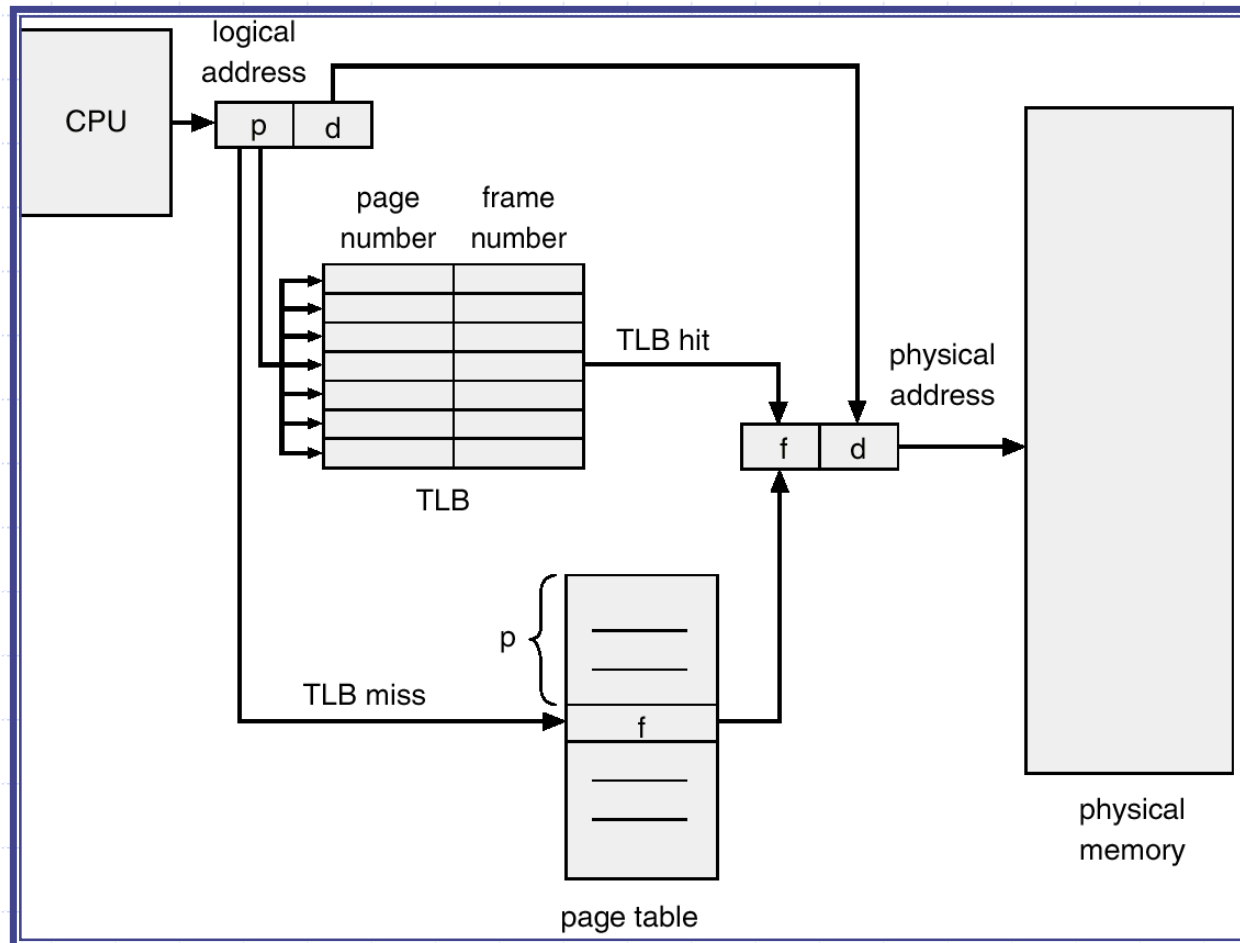
# Free Frames



# 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 (PRLR)* 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)*

# Paging Hardware With TLB



# Effective Access Time

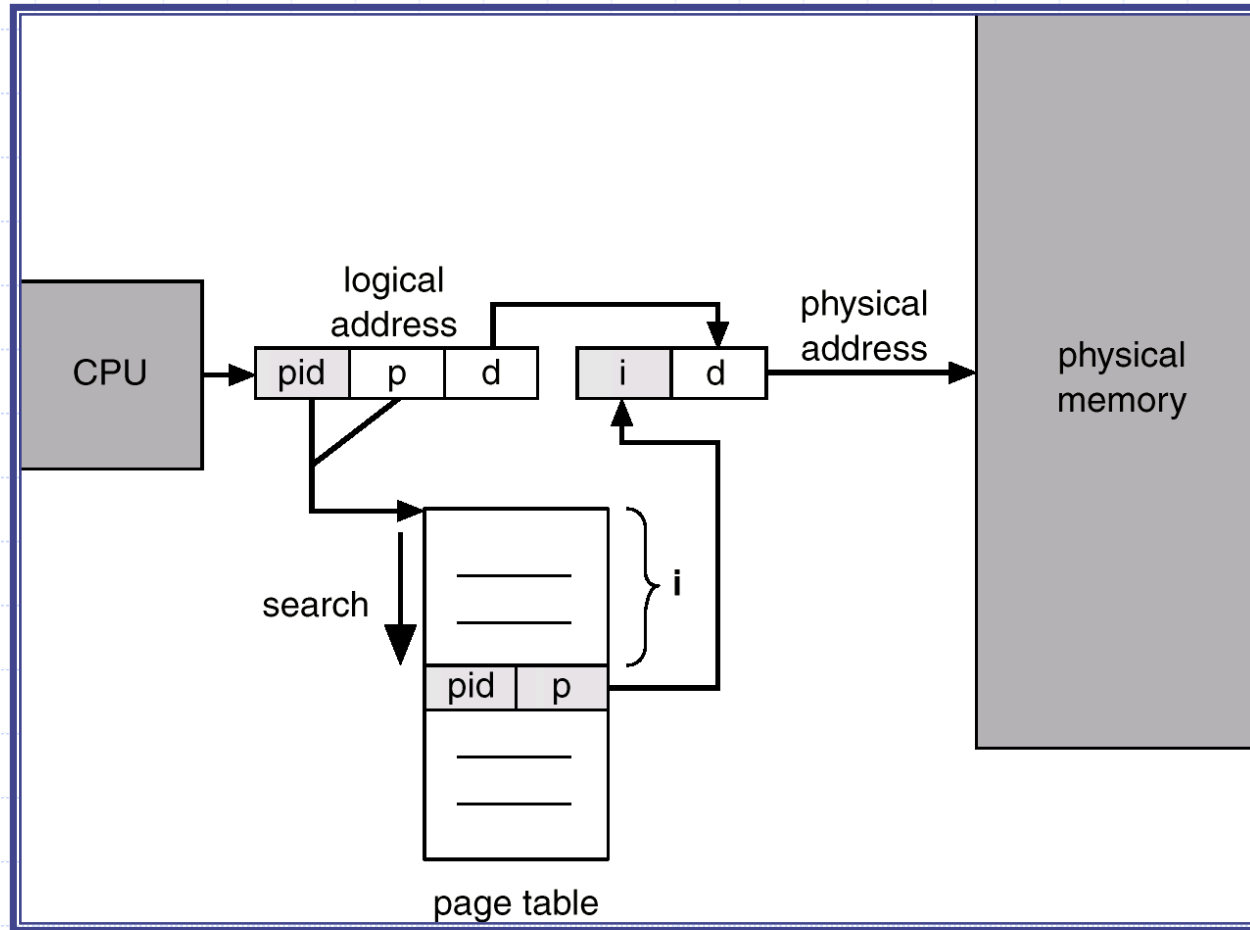
- ◆ Associative Lookup =  $\varepsilon$  time unit
- ◆ Assume memory cycle time is 1 microsecond
- ◆ Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- ◆ Hit ratio =  $\alpha$
- ◆ Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

# Inverted Page Table

- ◆ 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.

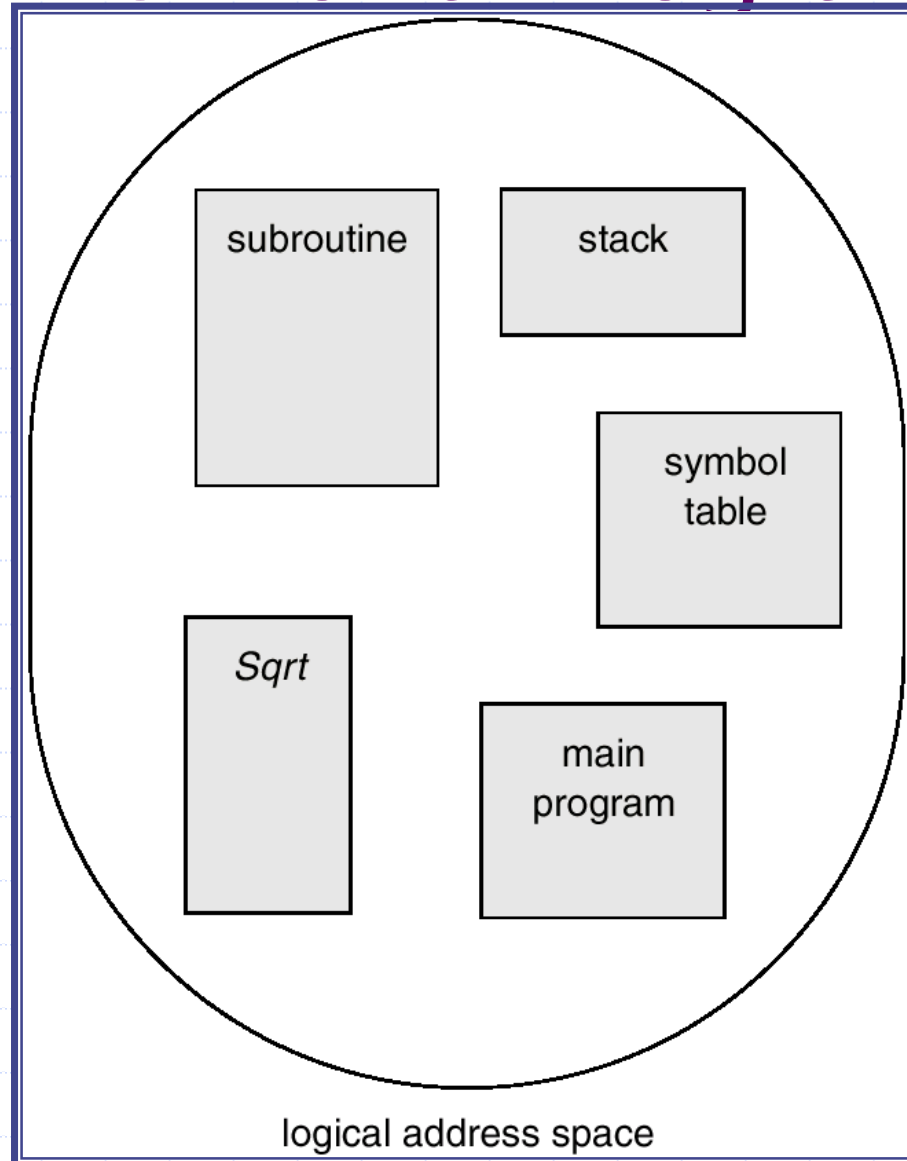
# Inverted Page Table Architecture



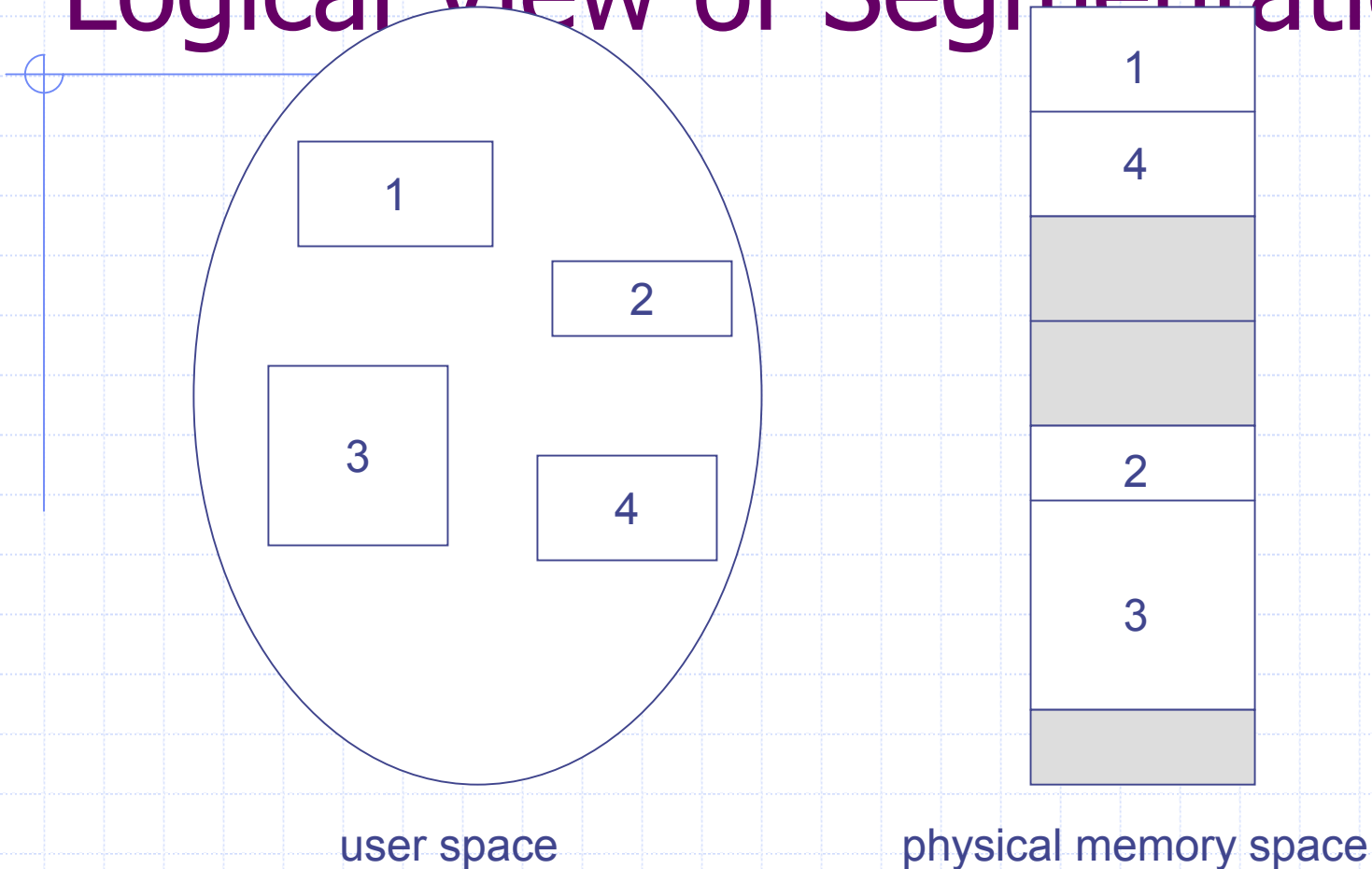
# Segmentation

- ◆ 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  $s < \text{STLR}$ .

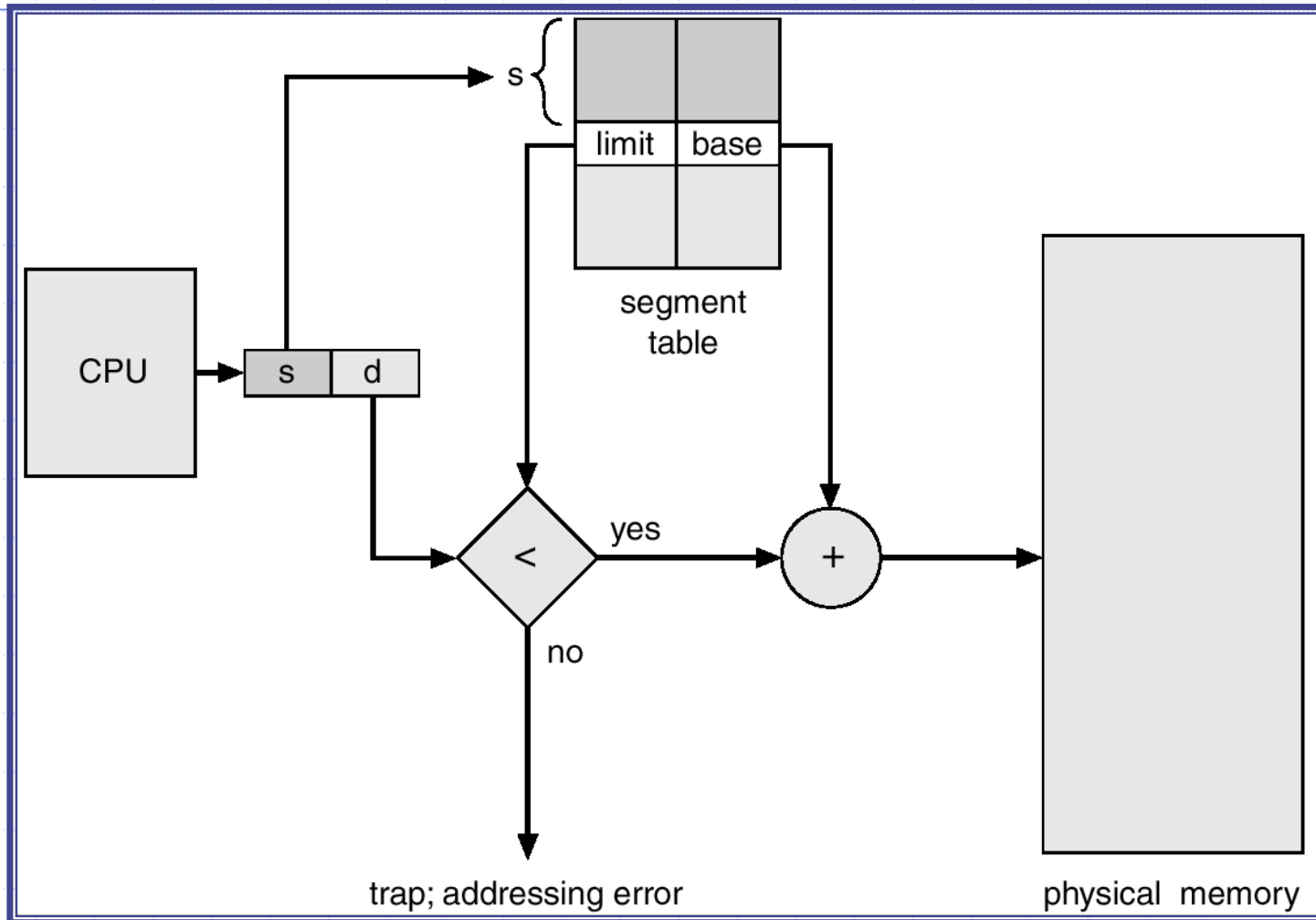
# Segmentation Architecture (Cont.)

- ◆ Relocation.
  - dynamic
  - by segment table
- ◆ Sharing.
  - shared segments
  - same segment number
- ◆ Allocation.
  - first fit/best fit
  - external fragmentation

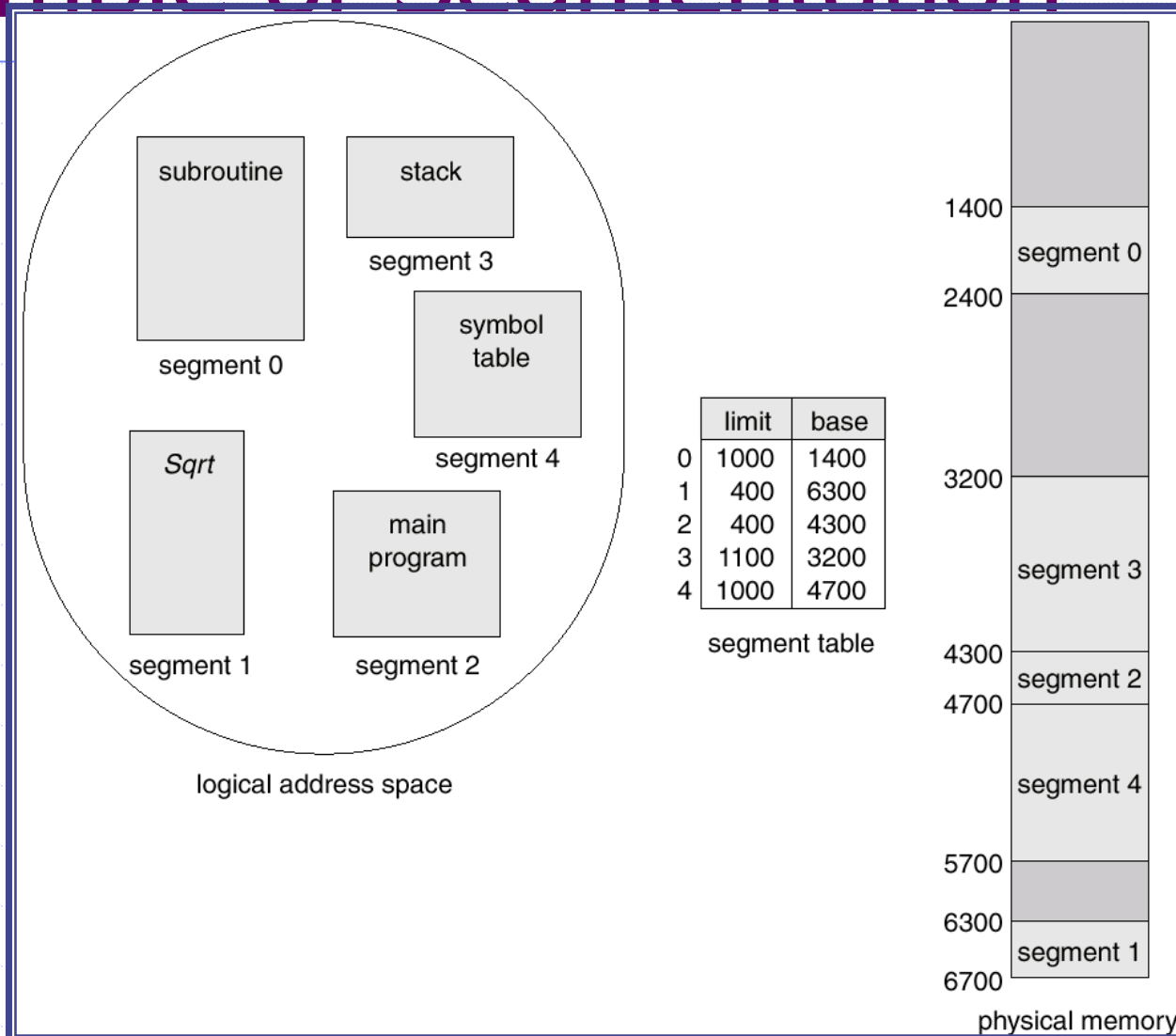
# Segmentation Architecture (Cont.)

- ◆ Protection. With each entry in segment table associate:
  - validation bit = 0  $\Rightarrow$  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



# Example of Segmentation



# Sharing of Segments

