Chapter 8 Main Memory

Contents

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium

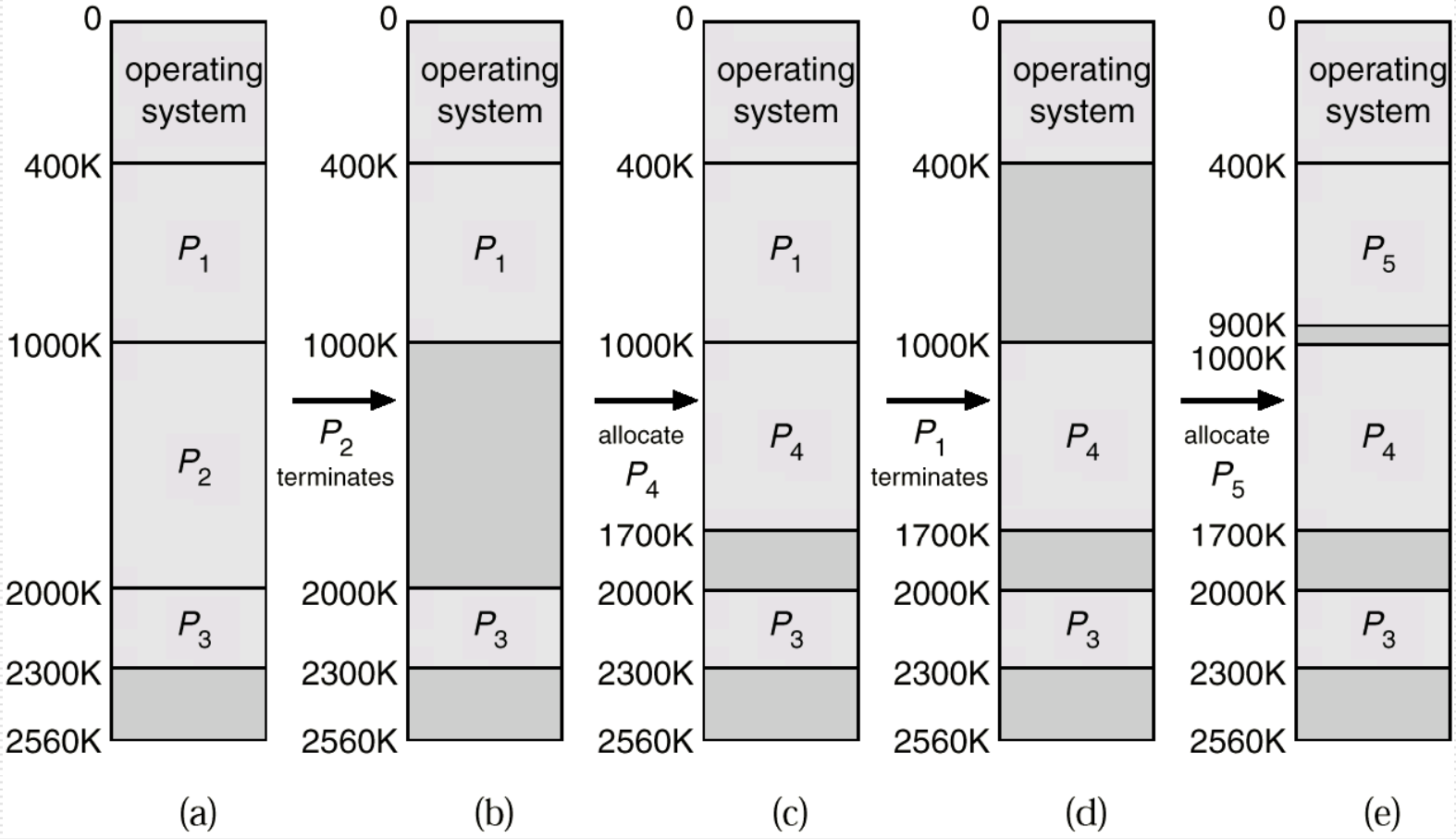
Objectives

- 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

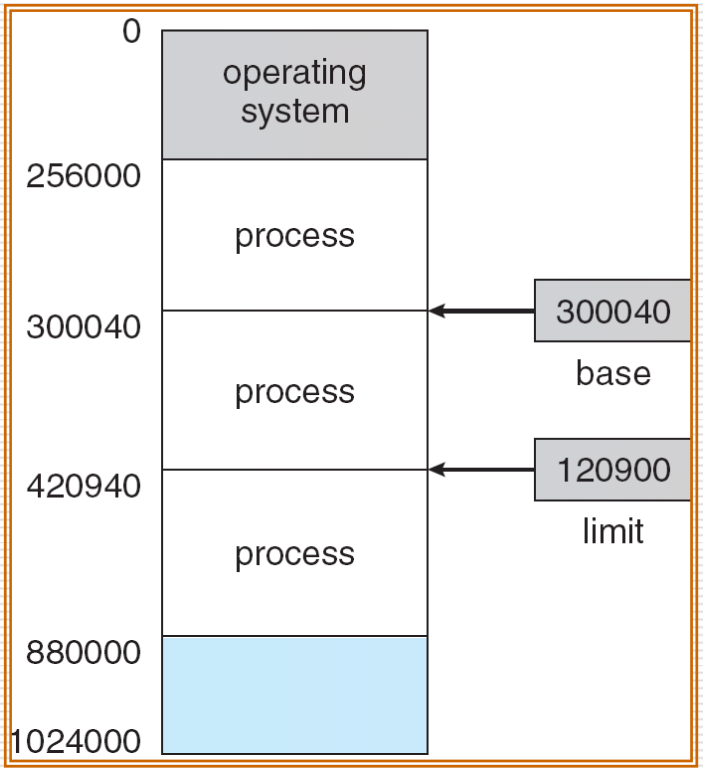
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Dynamic memory allocation

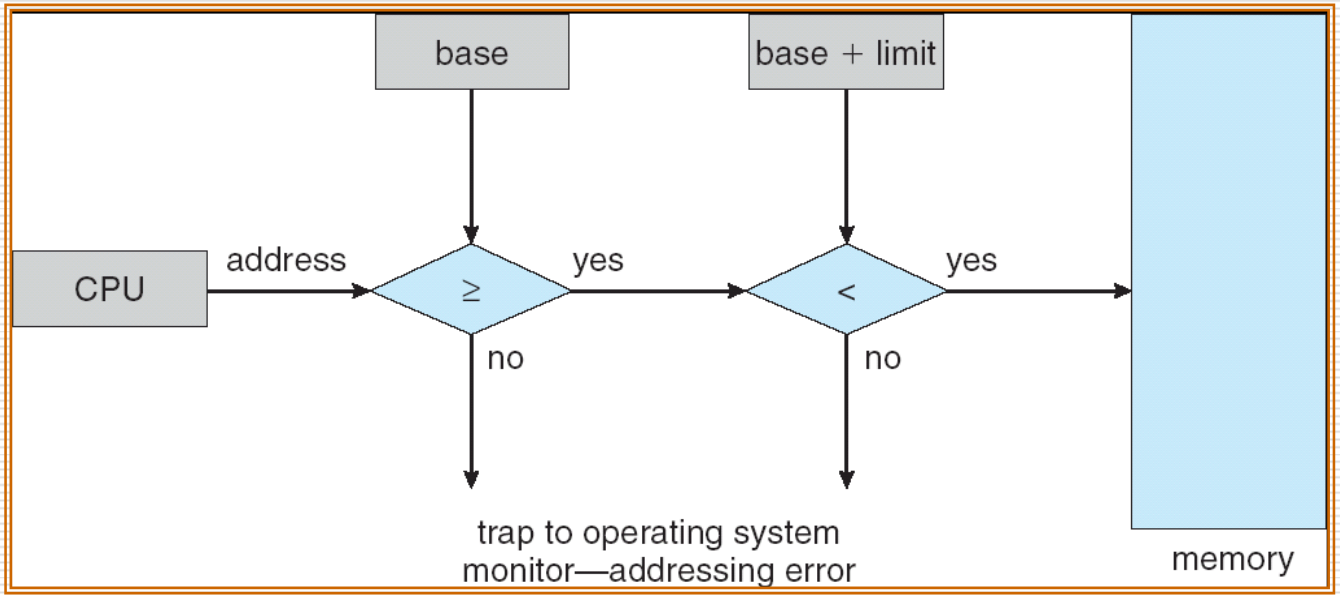


Base and Limit Registers

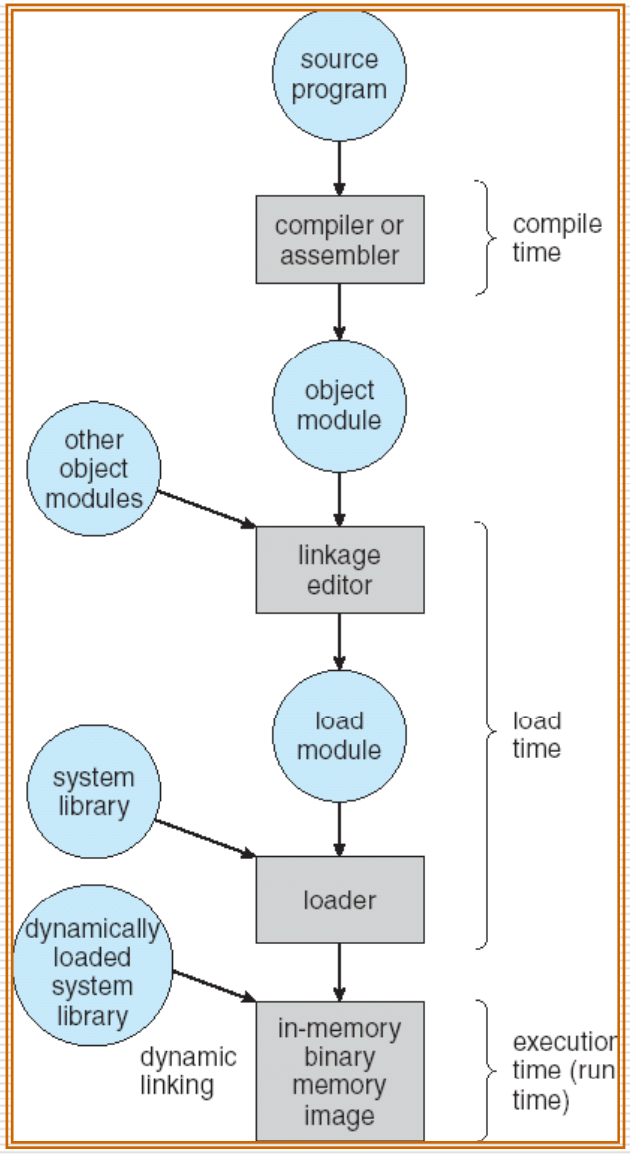
- A pair of **base** and **limit** registers define the logical address space



HW address protection with base and limit registers



Multi-step Processing of a User Program

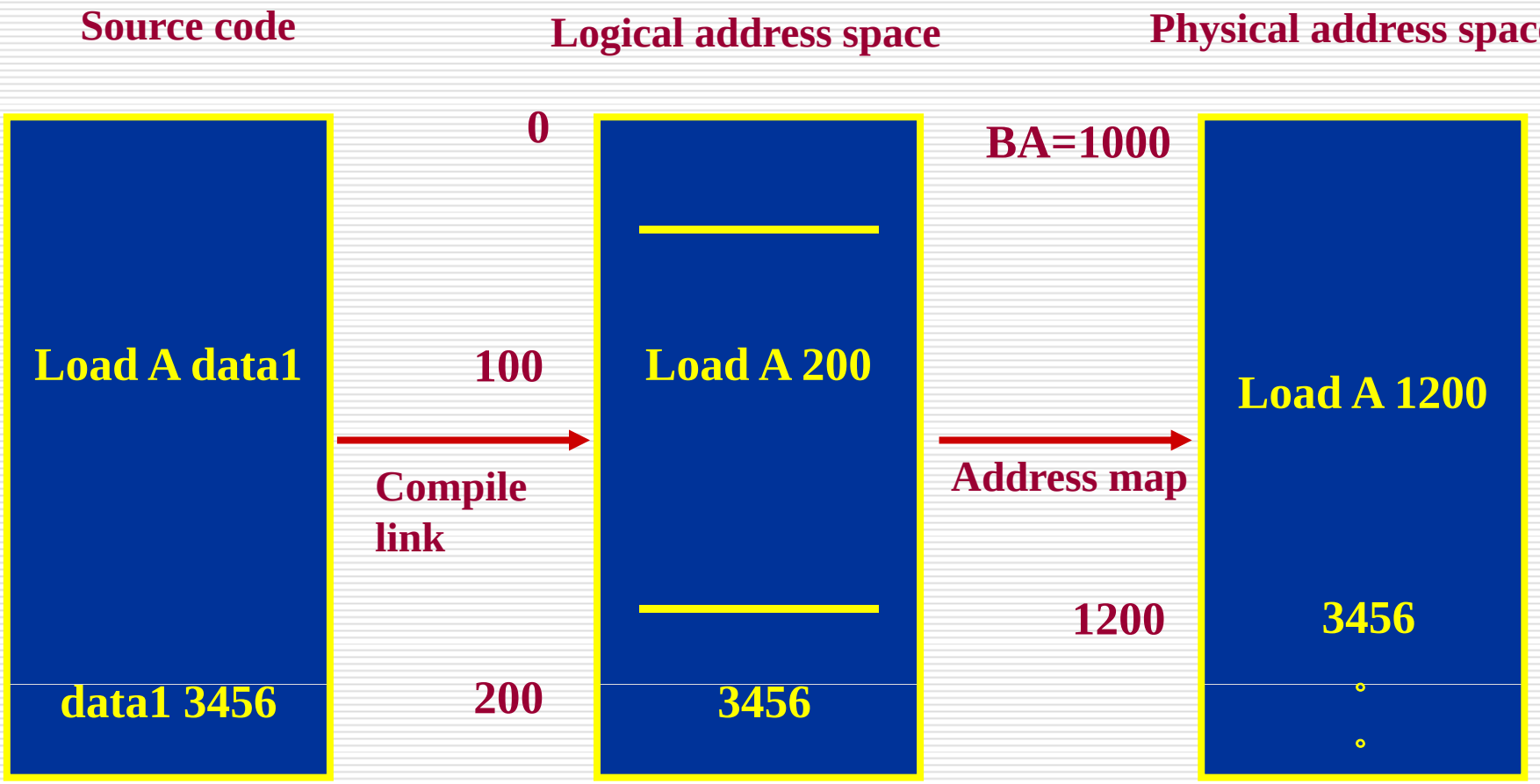


- In different step, address has different form
  - Symbol
  - Relocatable address
  - Absolute address

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**: 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)

Logical and physical address



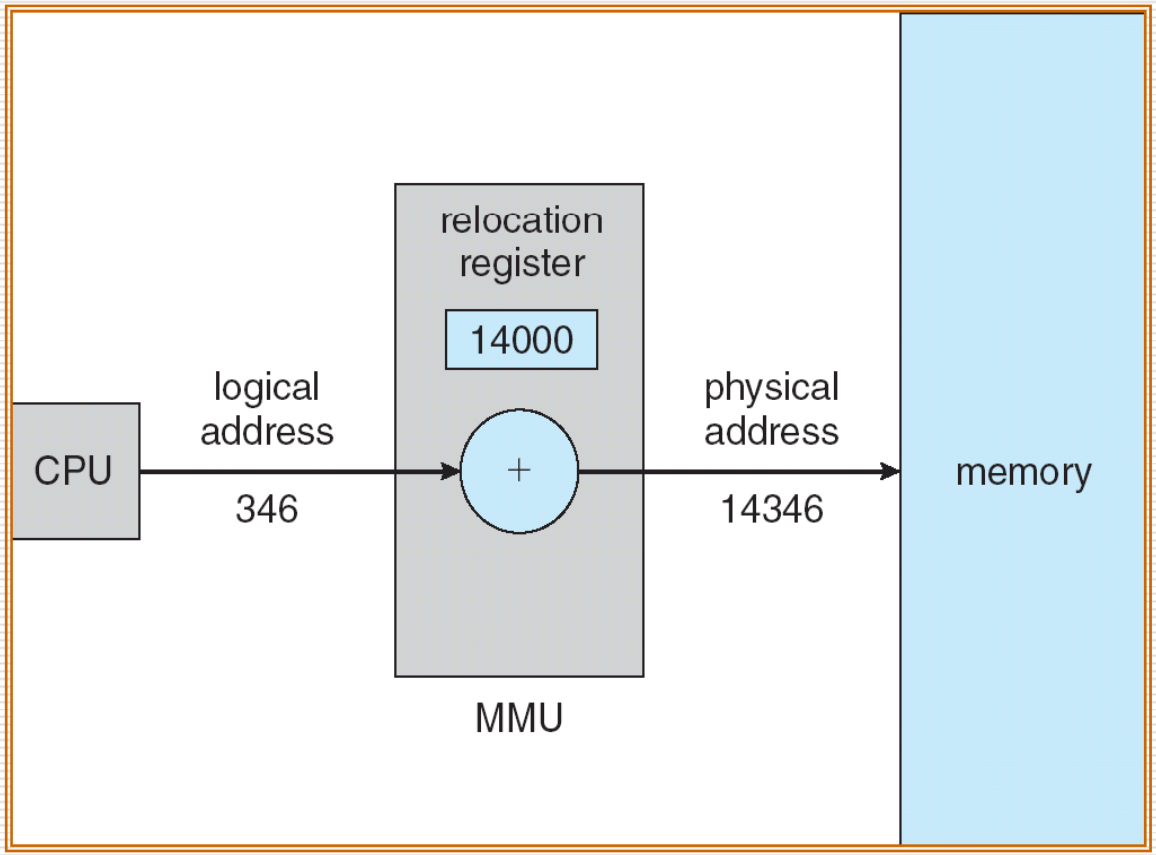
Logical vs. Physical Address Space

- 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



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

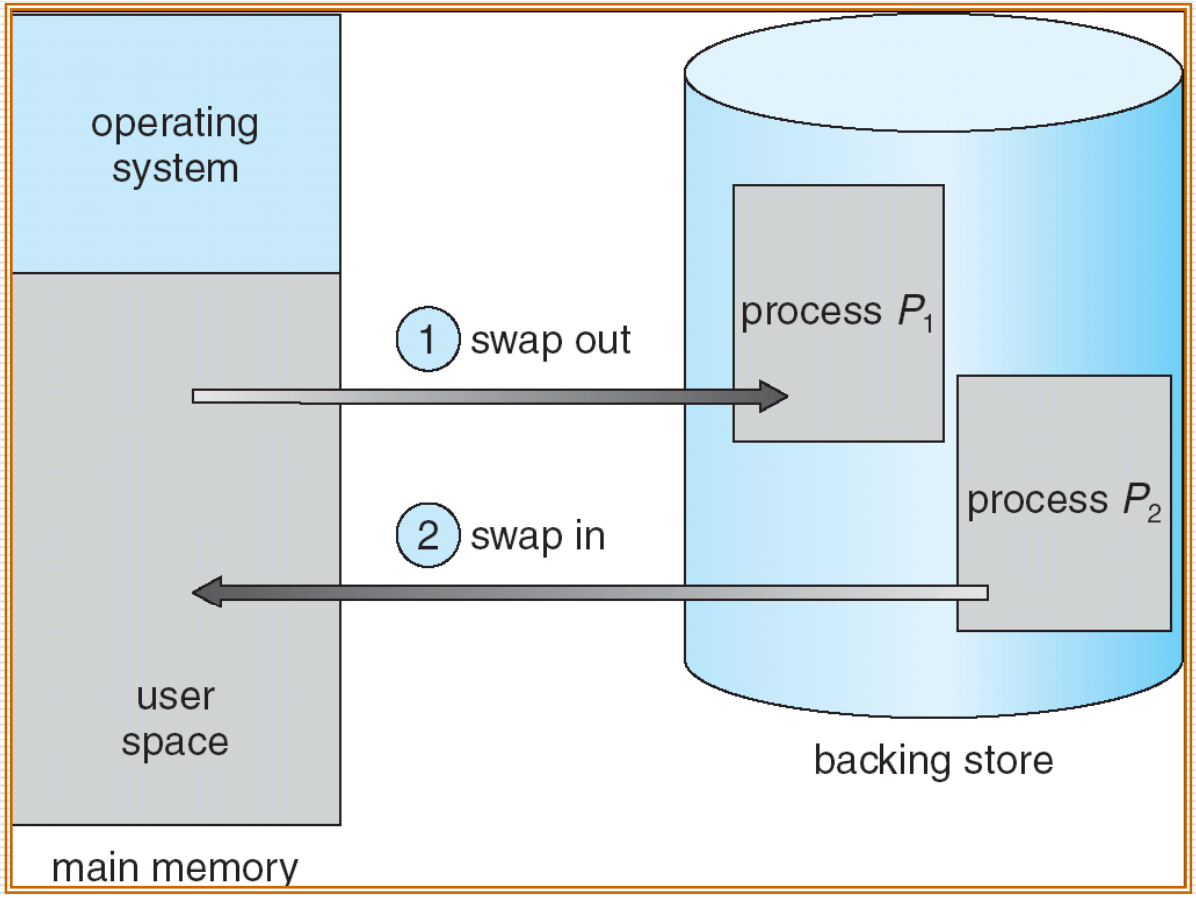
Dynamic Linking

- 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
- It can be extended to library updates.
- System also known as **shared libraries**

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



Problems in swapping

- When to swap
- Which to be swapped
- Where to be swapped in
  - Affected by bounding technology
- Other factors
  - A process to be swapped must be idle

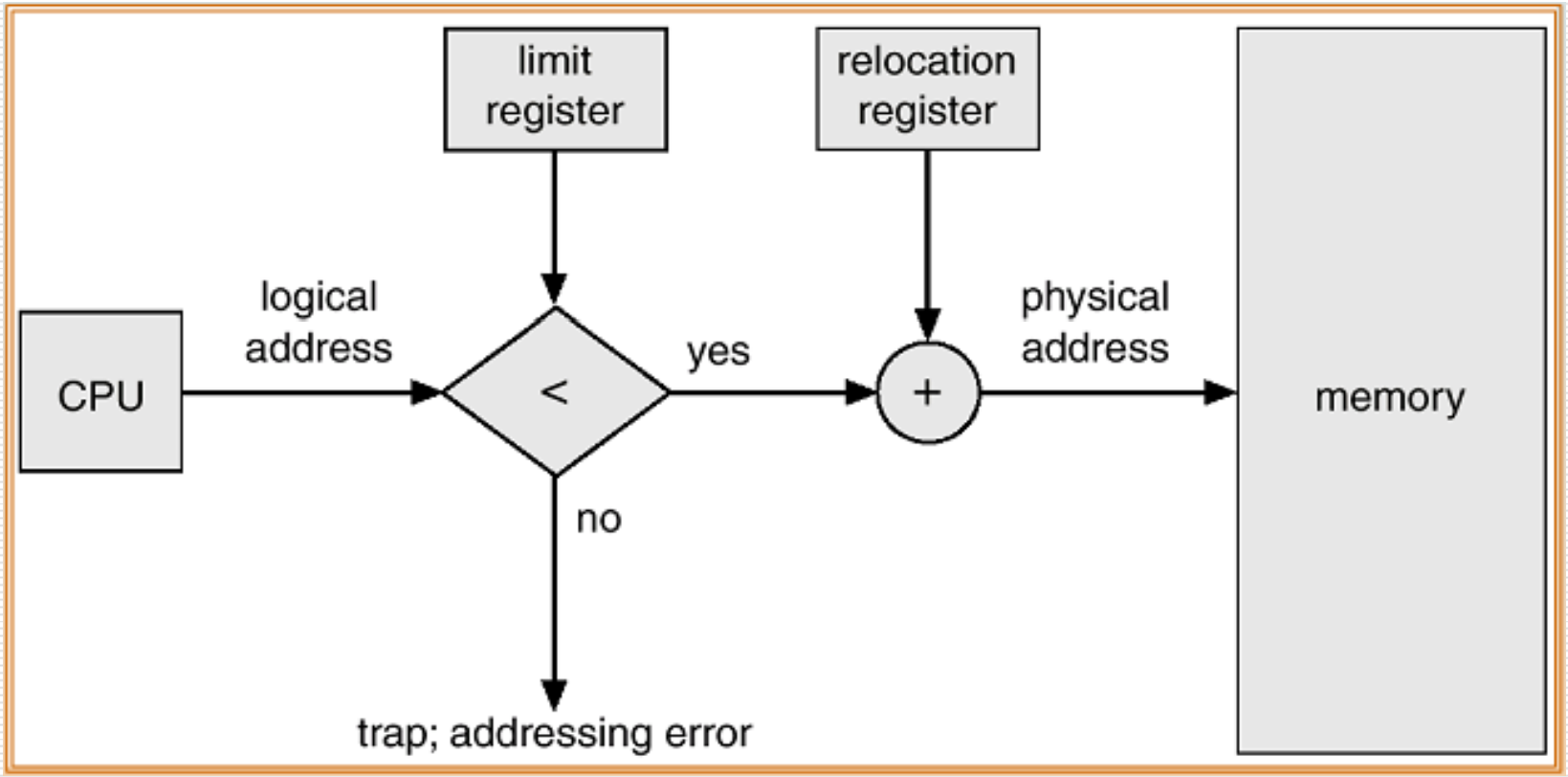
Swap time should be considered.

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*

Contiguous Allocation

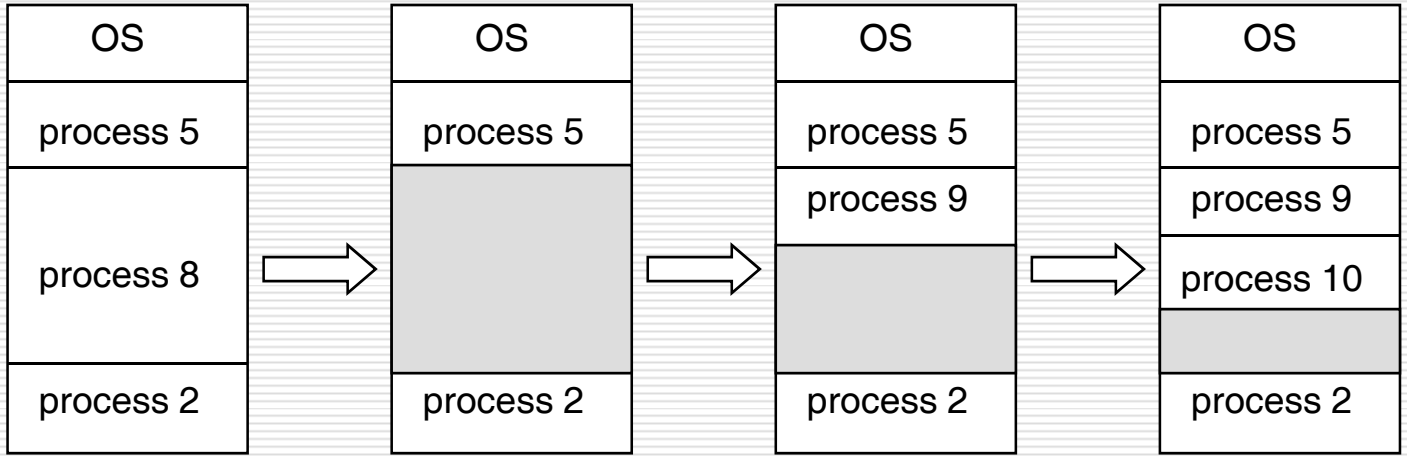


Contiguous Allocation (Cont.)

- 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)

Contiguous Allocation (Cont.)

- Dynamic allocation scheme

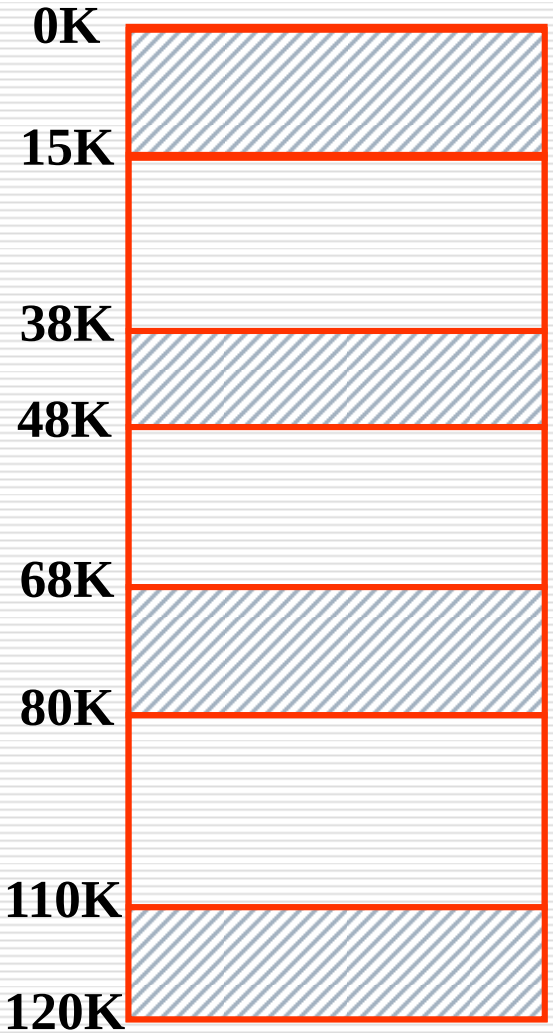


Dynamic Storage-Allocation Problem

- 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

Example



始址	长度	标志
15K	23K	未分配
48K	20K	未分配
80K	30K	未分配
		空
		空

已分配区表

始址	长度	标志
0K	15K	J1
38K	10K	J2
68K	12K	J3
110K	10K	J4
		空
		空

Suppose that a program needs 19k

Fragmentation

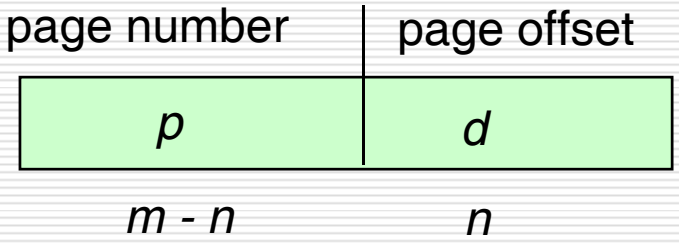
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation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

- physical 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
- Internal fragmentation

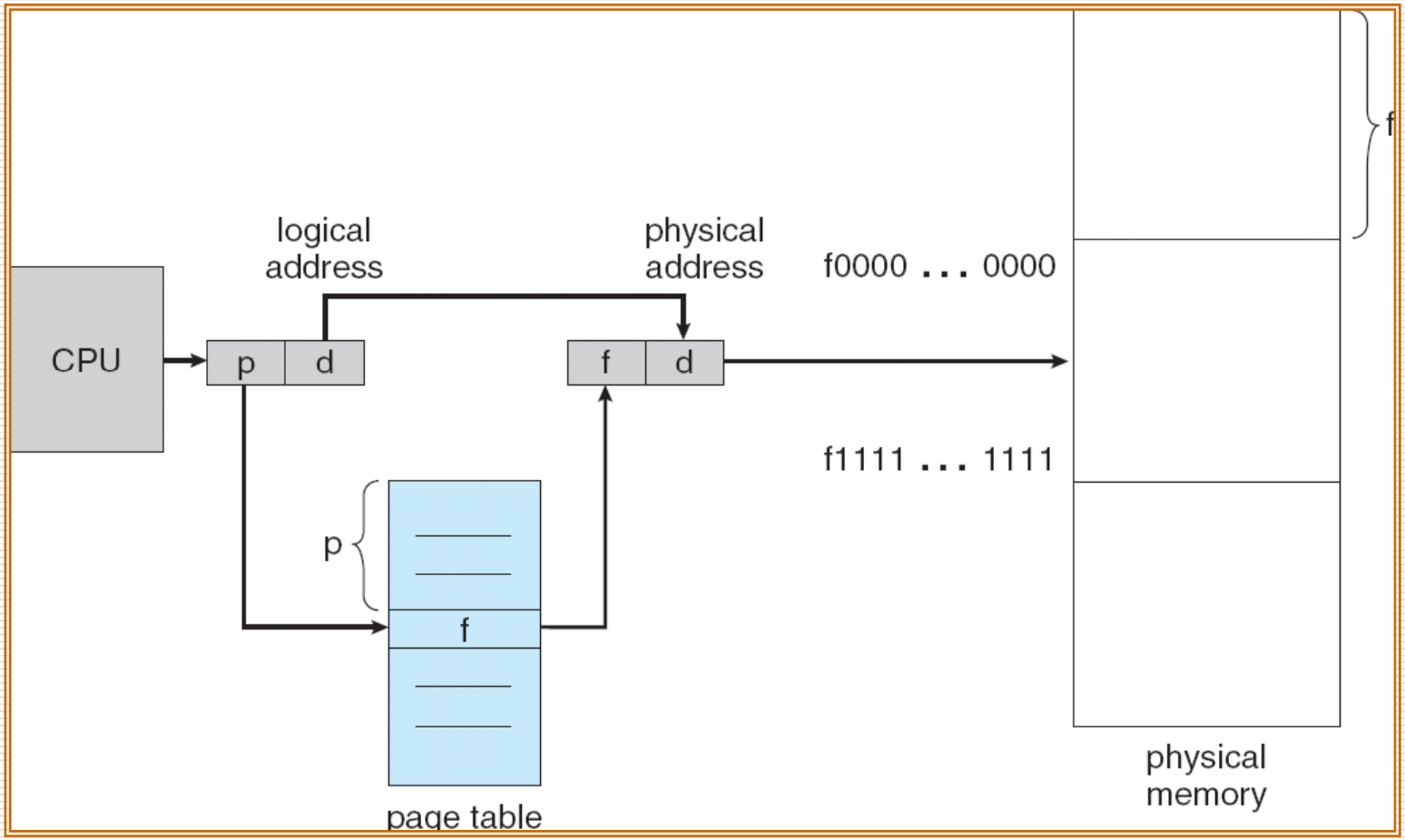
Address Translation Scheme

- 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

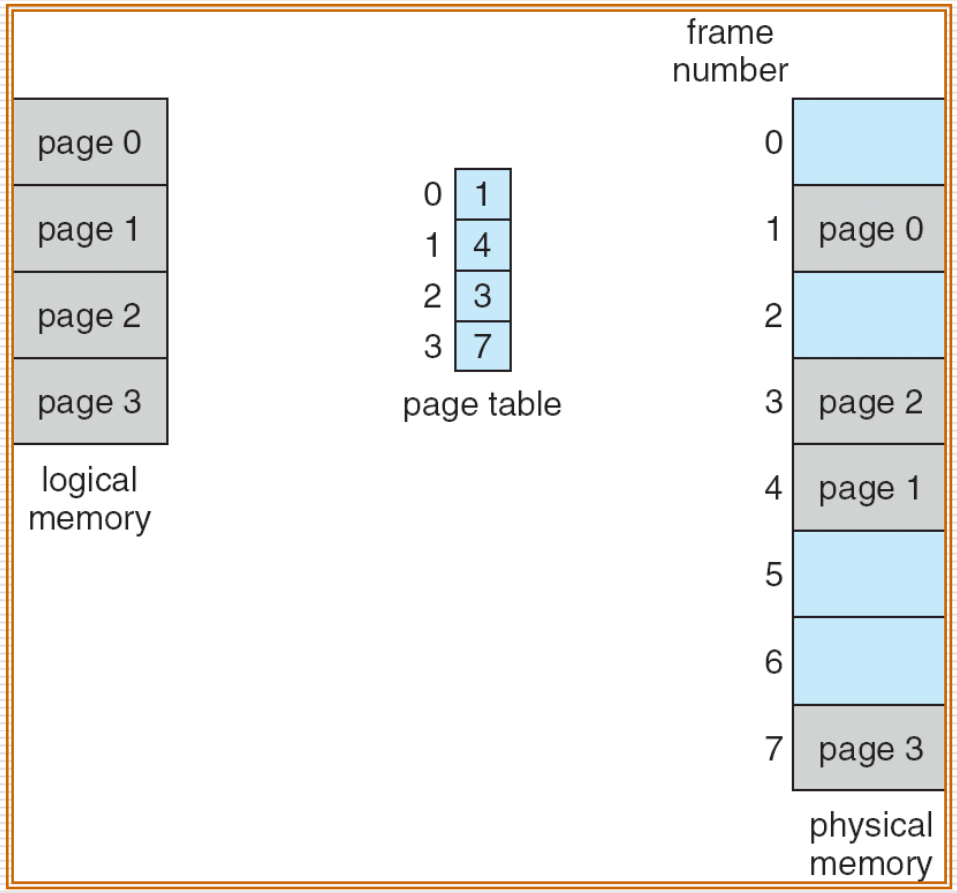


  - For given logical address space 2^m *and page size* 2^n

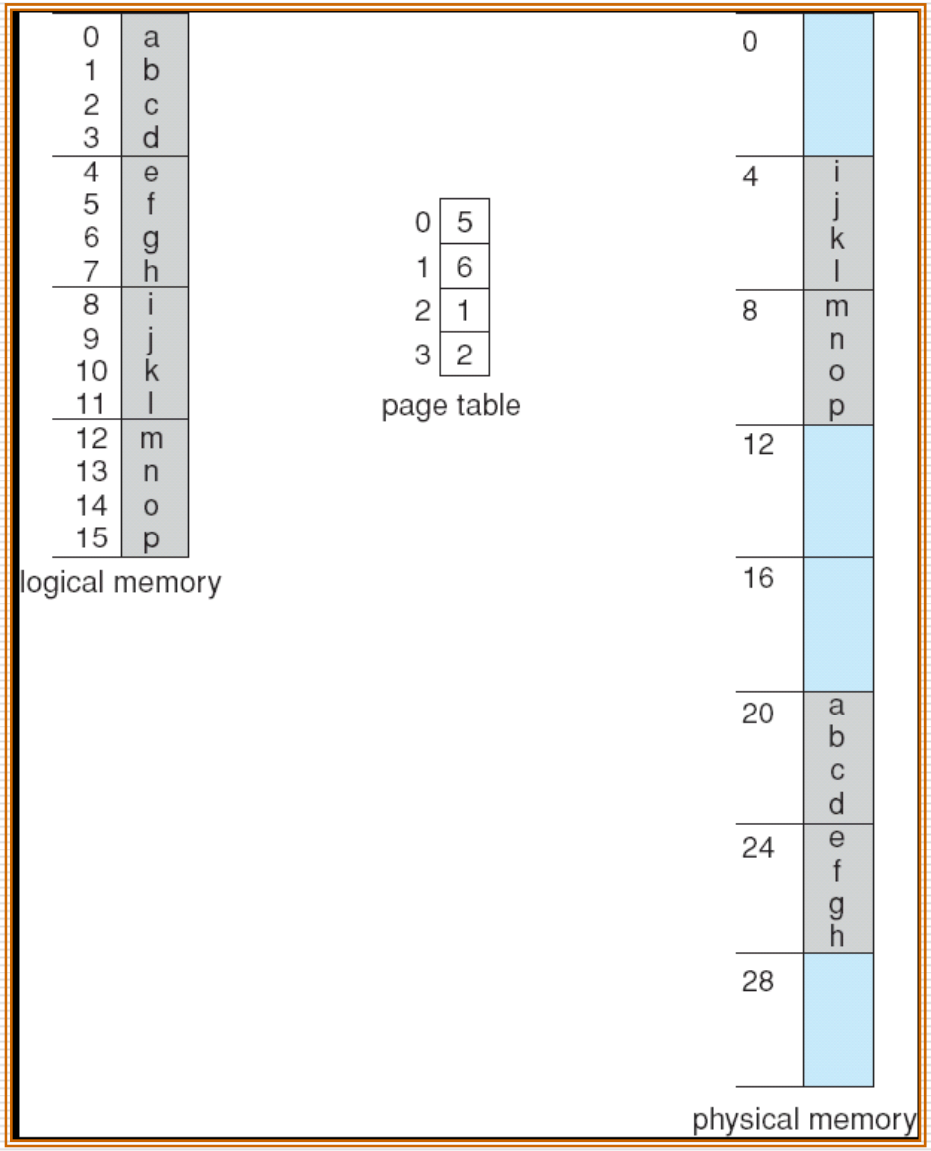
Paging Hardware



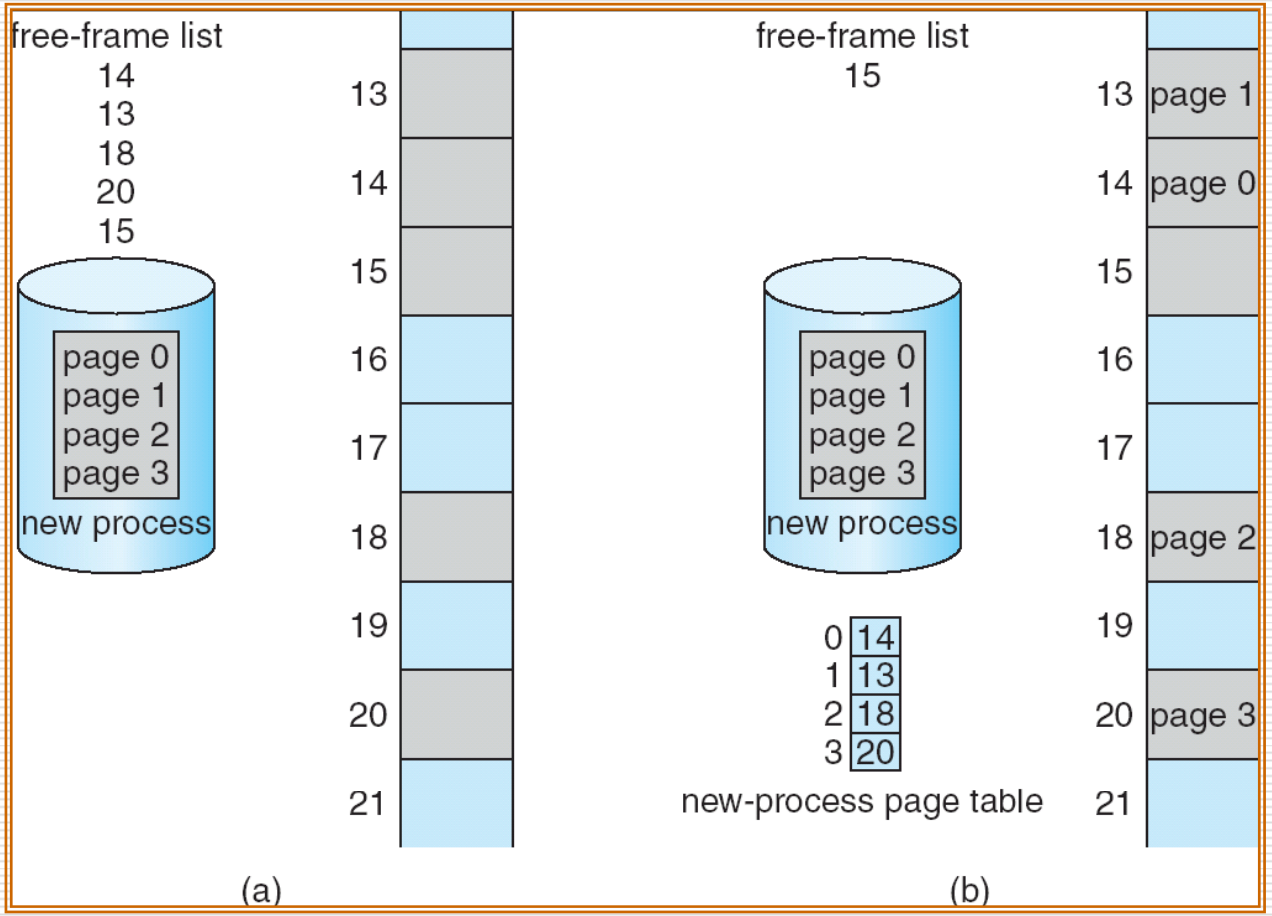
Paging Model of Logical and Physical Memory



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)

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)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

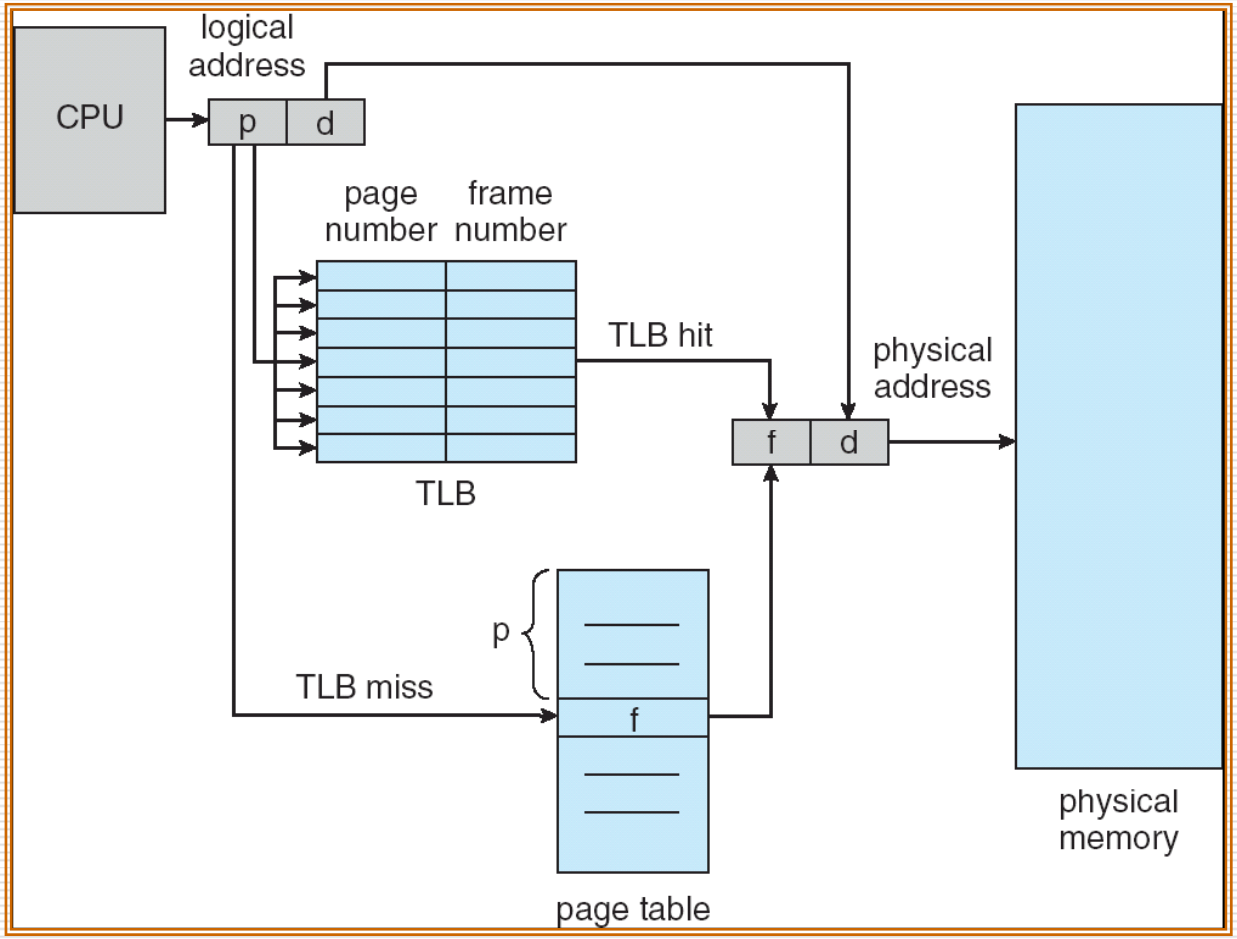
Associative Memory

- Associative memory – parallel search

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

- 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)

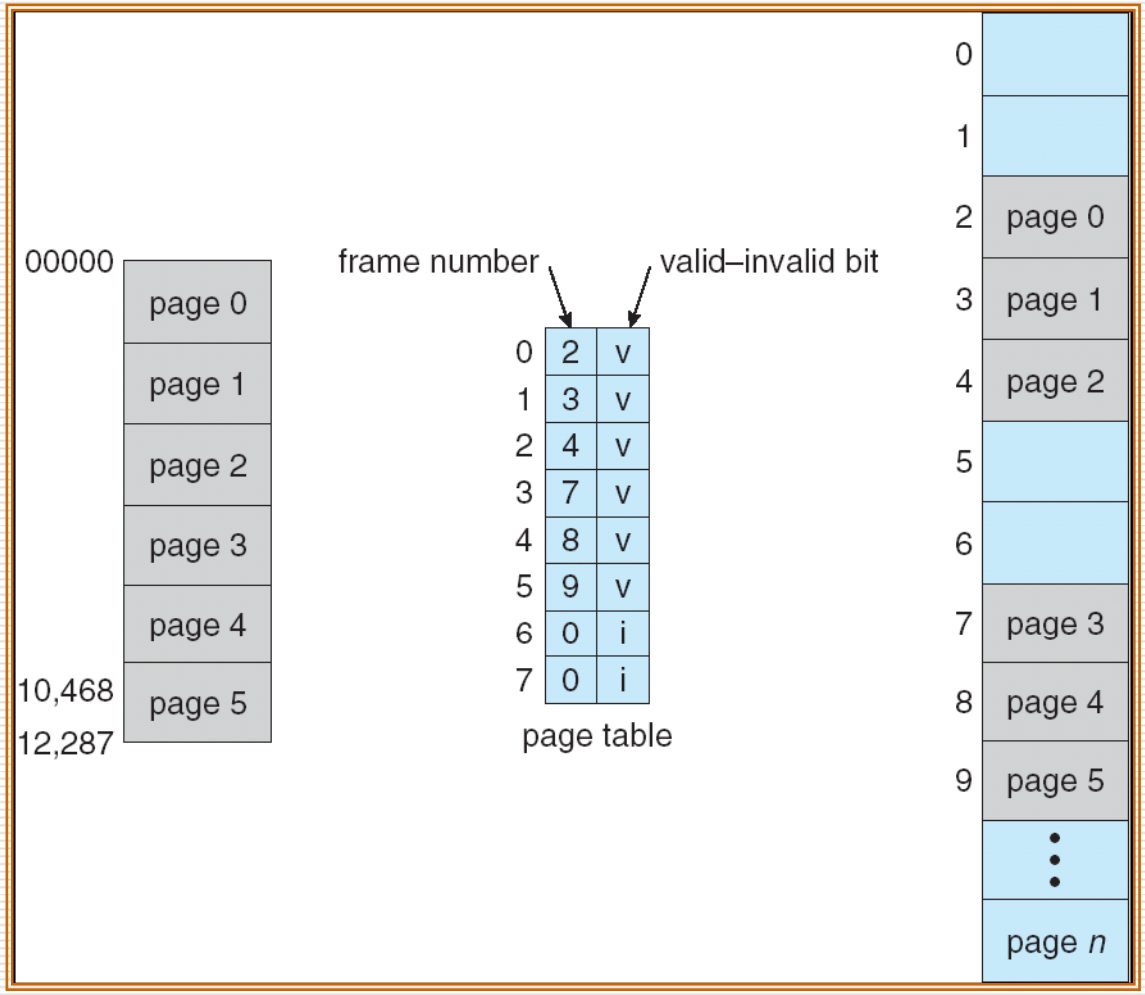
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Memory Protection

- 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

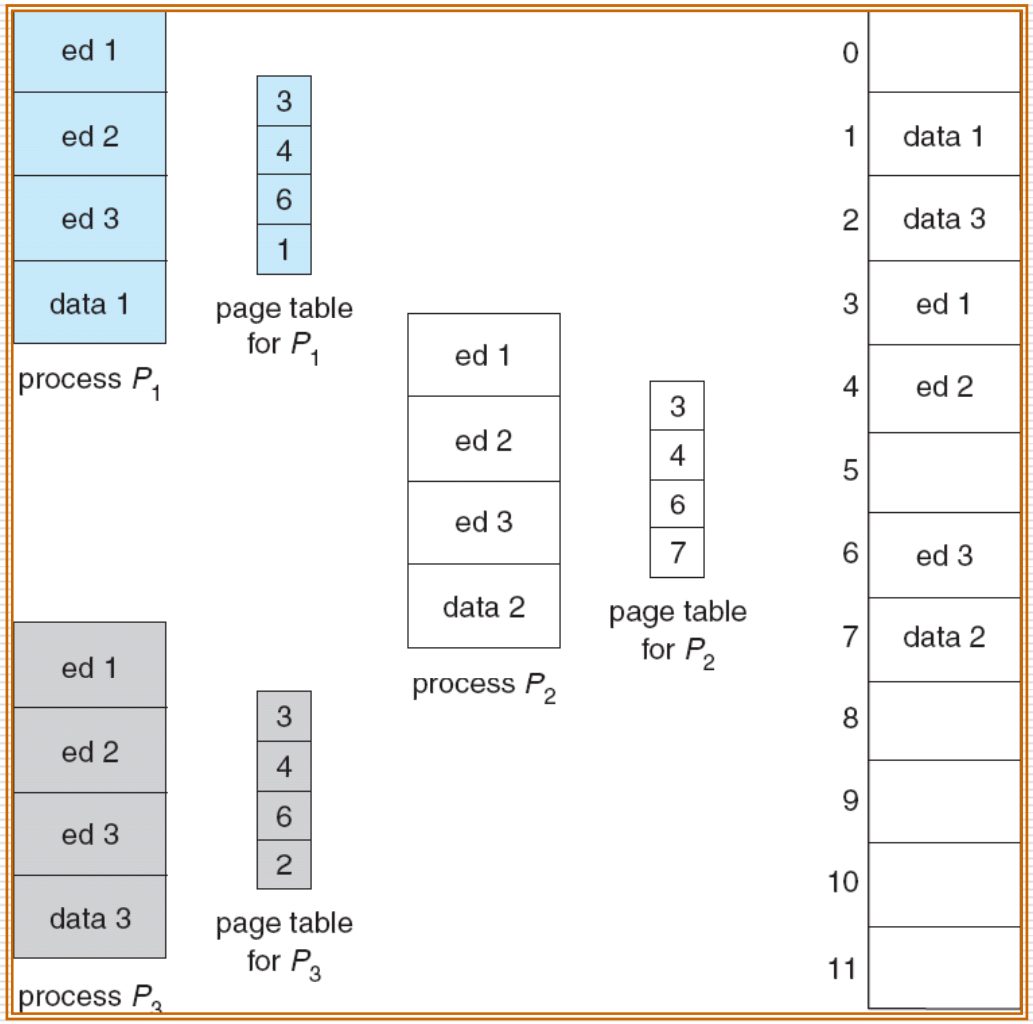
Valid (v) or Invalid (i) Bit In A Page Table



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 (To be specific, it is at least *same* offset from start of page)
- **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

- Problems of the contiguous page table

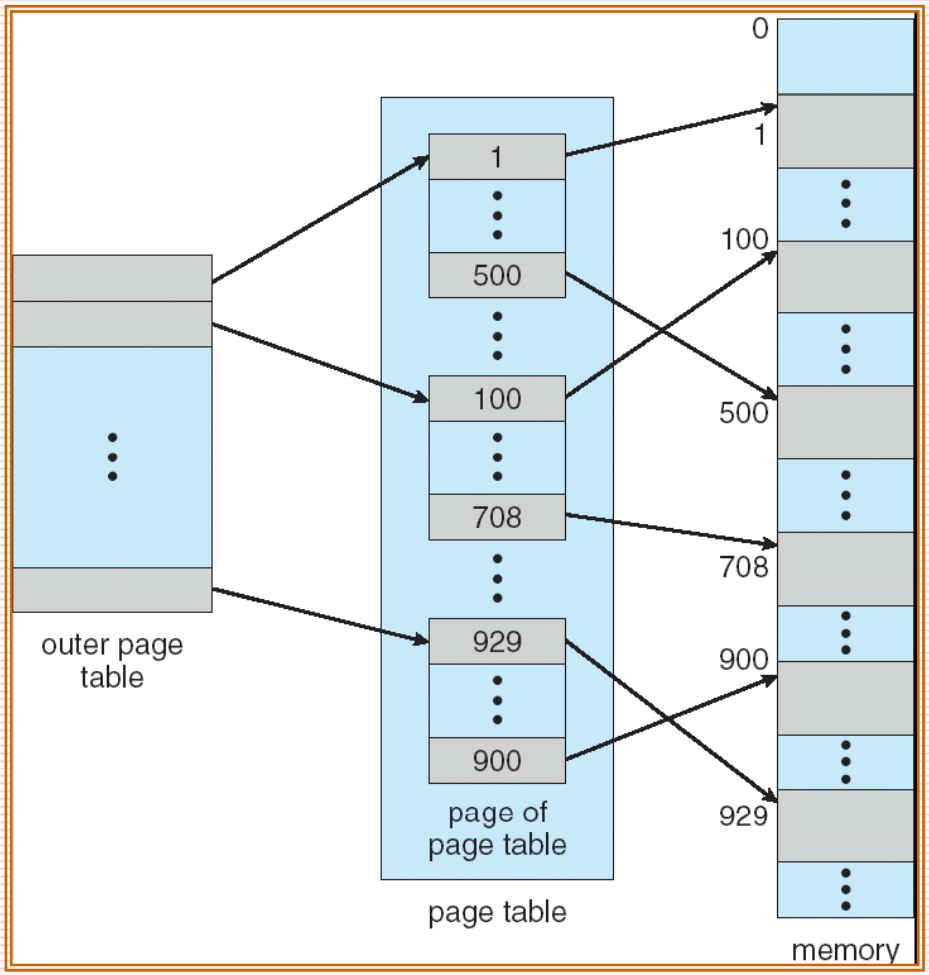
- Methods
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables

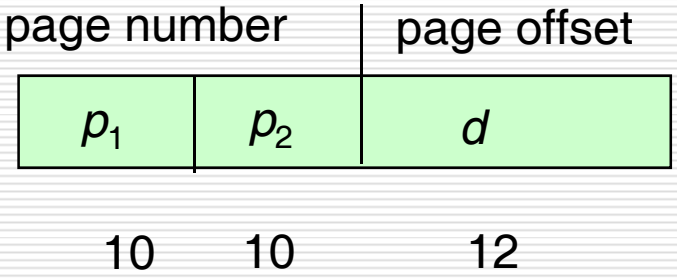
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



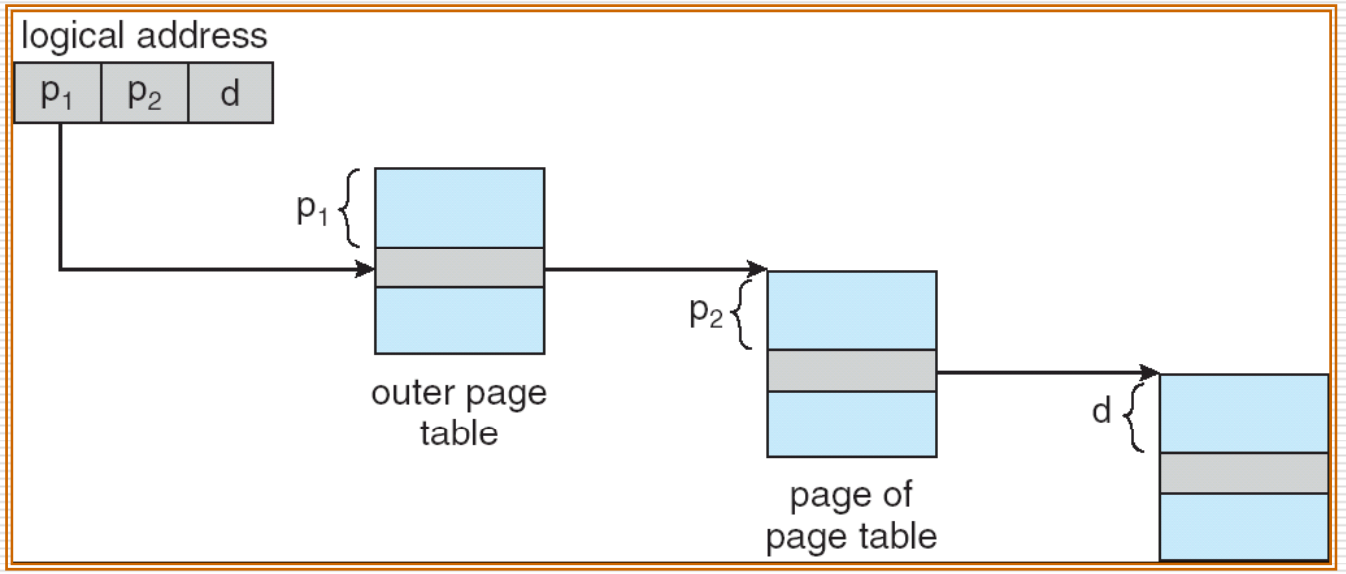
Two-Level Paging Example

- 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
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



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

Address-Translation Scheme



Three-level Paging Scheme

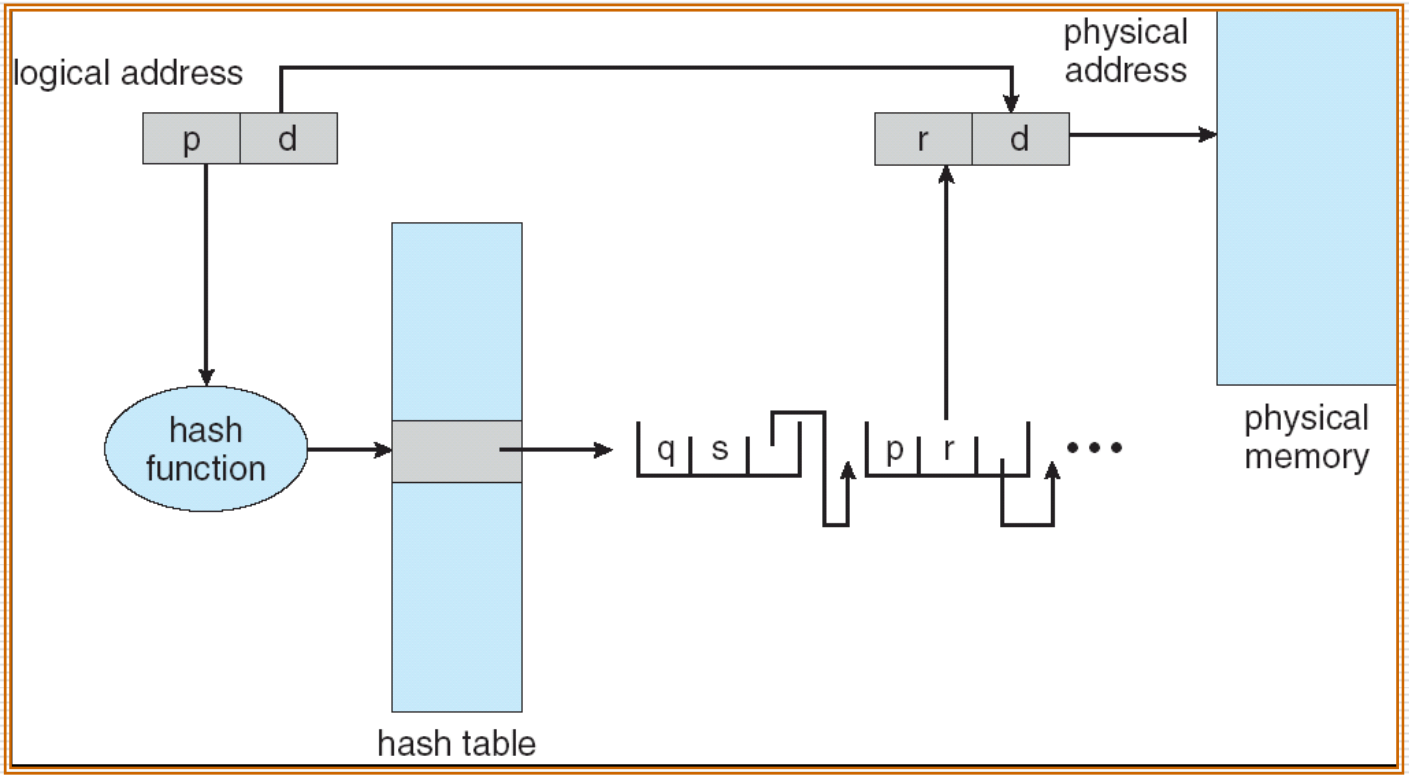
outer page	inner page	offset
p_1	p_2	d
42	10	12

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.

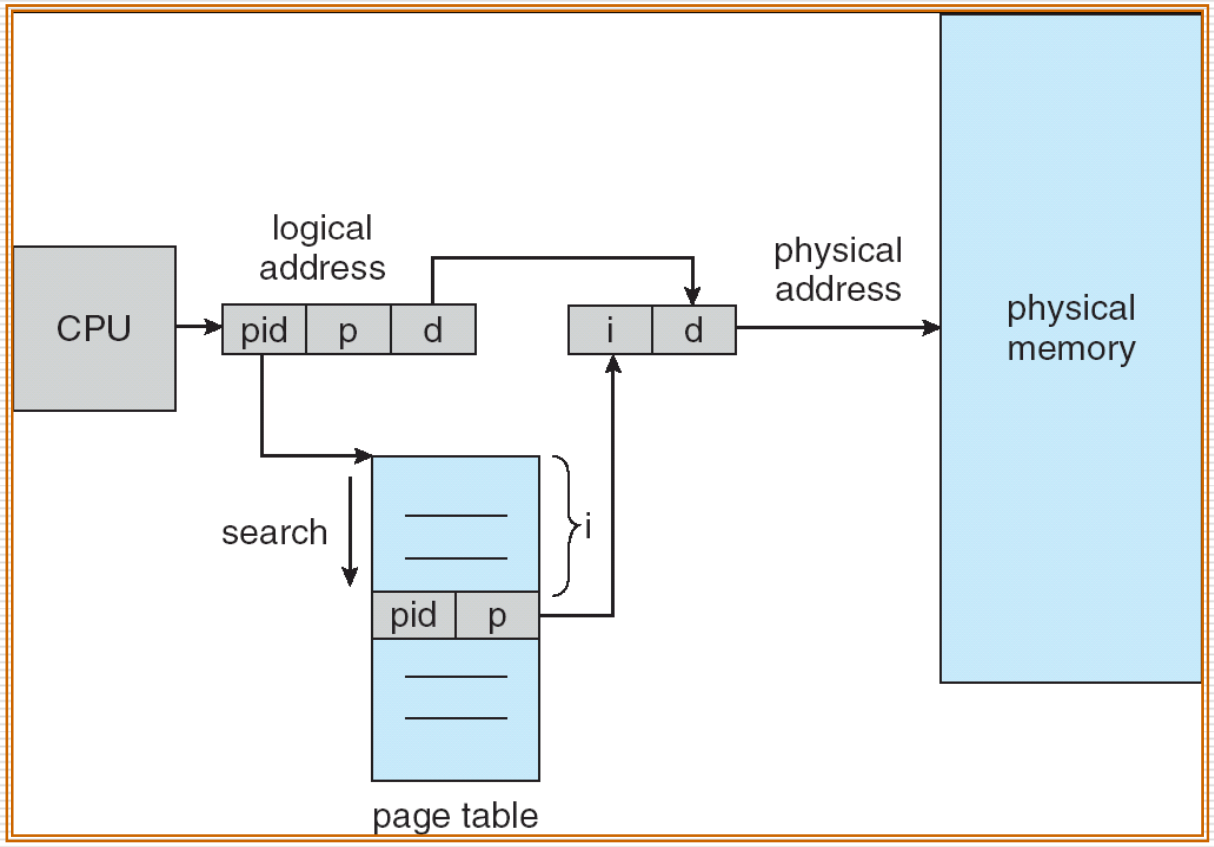
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- Each 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

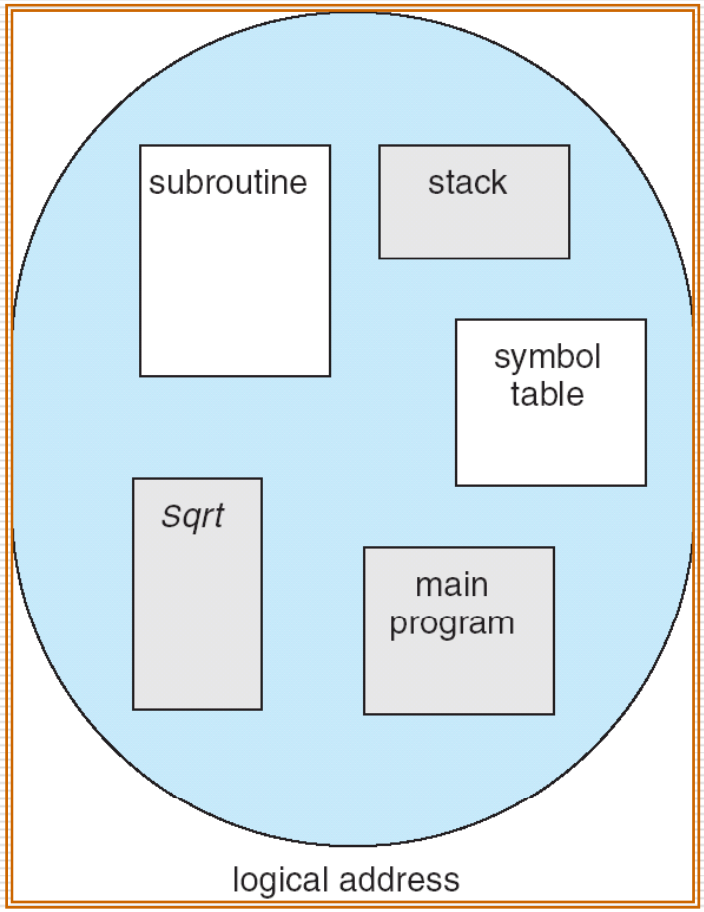
Inverted Page Table Architecture



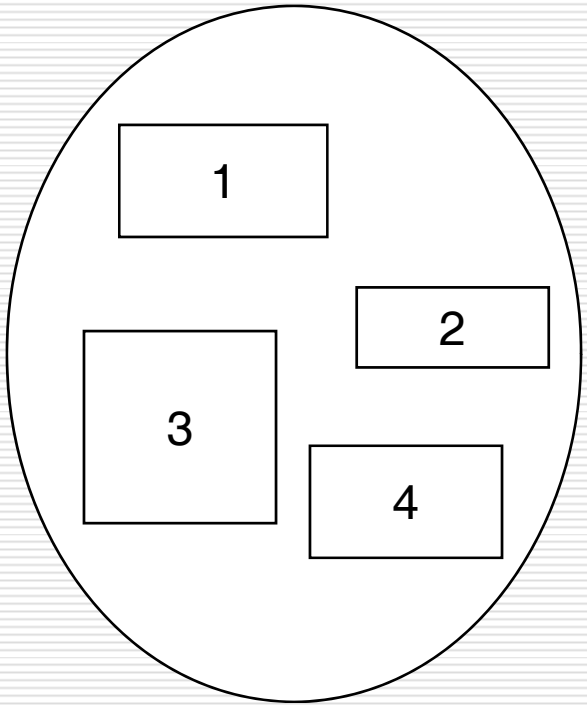
Segmentation

- 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

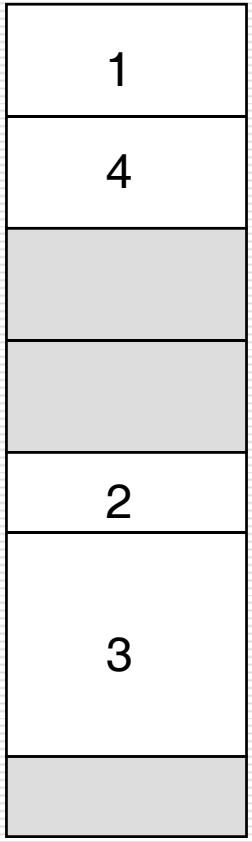
User's View of a Program



Logical View of Segmentation



user space



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

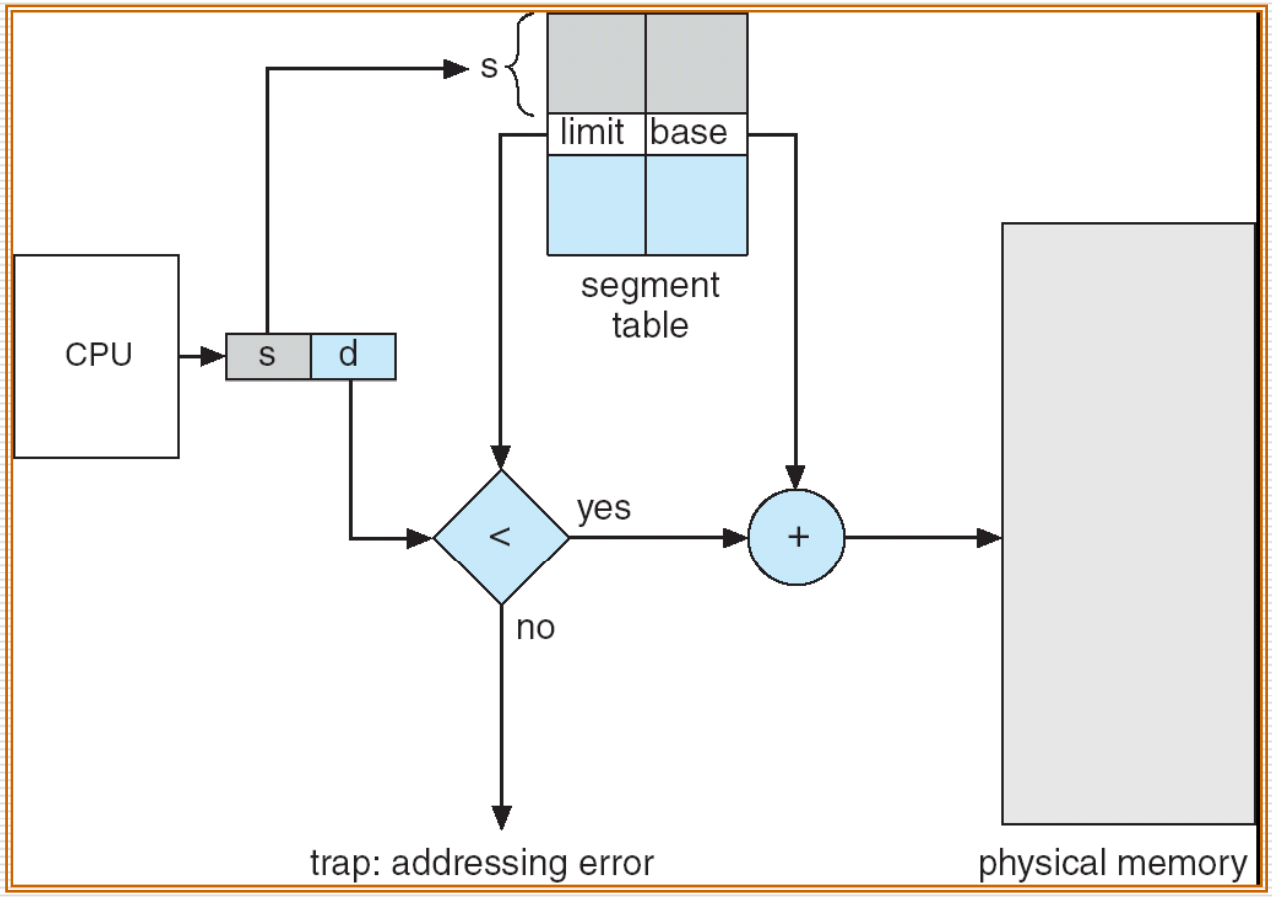
- **Segment table –** maps two-dimensional physical addresses; each table entry has:
  - **base –** contains the starting physical address where the segments reside in memory
  - **limit –** specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

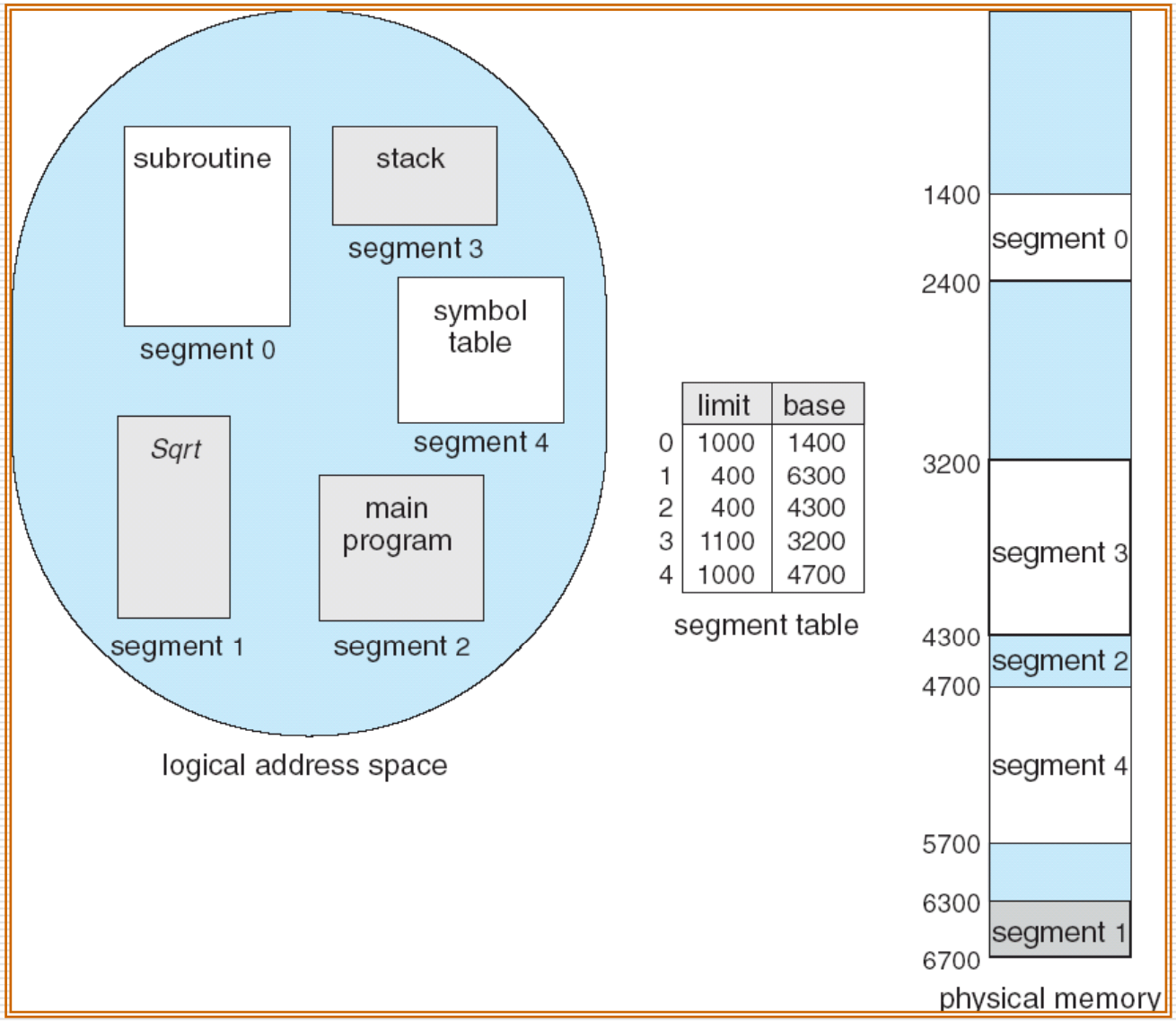
Segmentation Architecture (Cont.)

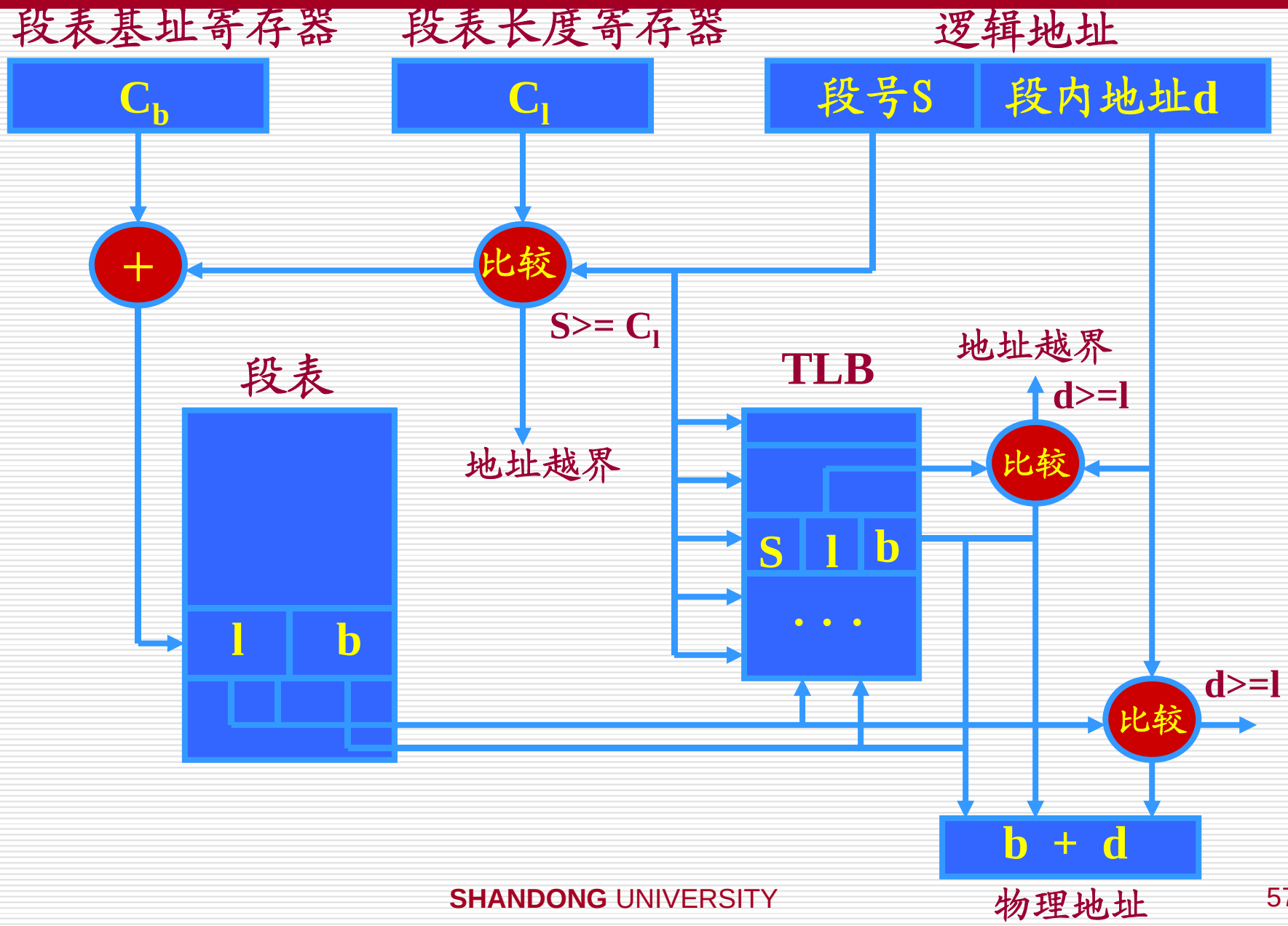
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation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

Segmentation Hardware



Example of Segmentation




段表基址寄存器
段表长度寄存器
逻辑地址
C_b
C_l
段号S
段内地址d
+
比较
S>= C_l
地址越界
段表
l
b
TLB
S
l
b
地址越界
d>=l
比较
比较
d>=l
b + d
物理地址

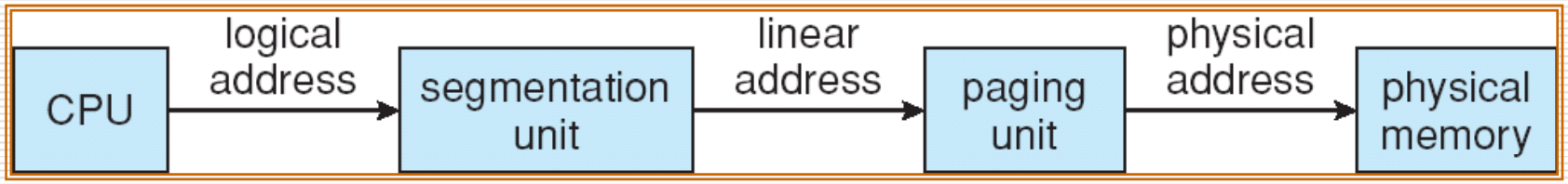
Segmentation vs. paging

- Similarity:
- Address space can exceed size of real memory.
- Differences:
- Programmer is aware of segmentation. Paging is hidden.
- Segmentation allows procedures and data to be *separately protected*. This is hard with paging.
- Segmentation easily permits tables whose size varies.
- Segmentation *facilitates sharing* of procedures between processes. This is hard with paging.
- *Pure* segmentation suffers from *memory fragmentation*.

Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

Logical to Physical Address Translation in Pentium

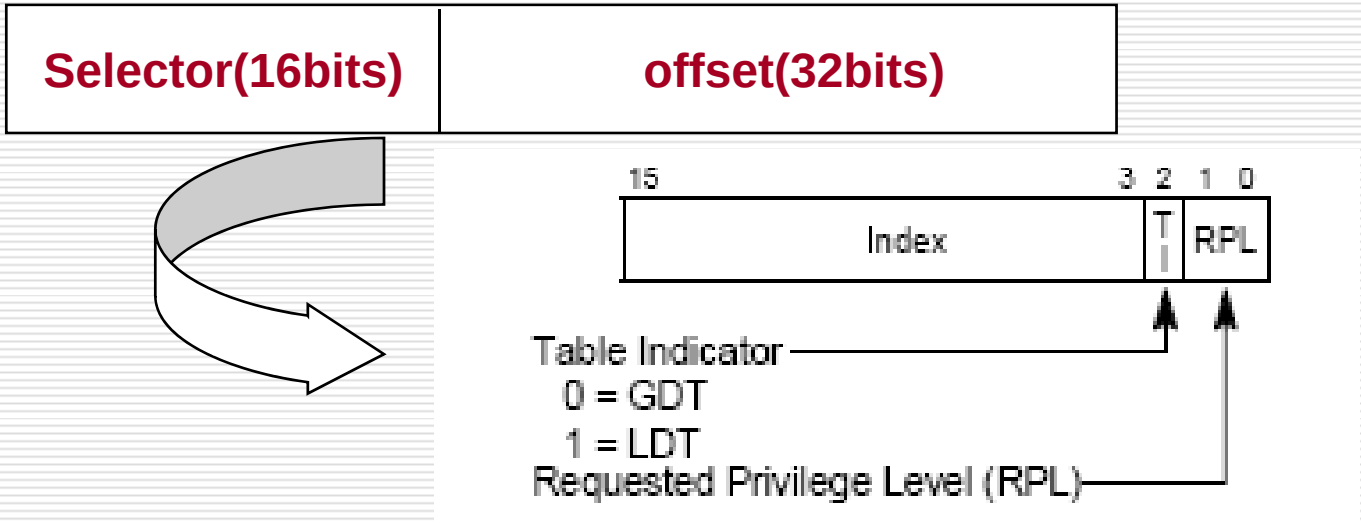


Intel Pentium Segmentation

- X86 Processor has Several segment registers
  - CS (code segment)
  - DS (Data Segment)
  - SS (Stack segment)
  - ES (Extra segment)
  - GS & FS segment
- Pentium allows a segment to be as large as 4 GB
- Selector is 16 bit
- Offset is 32 bit

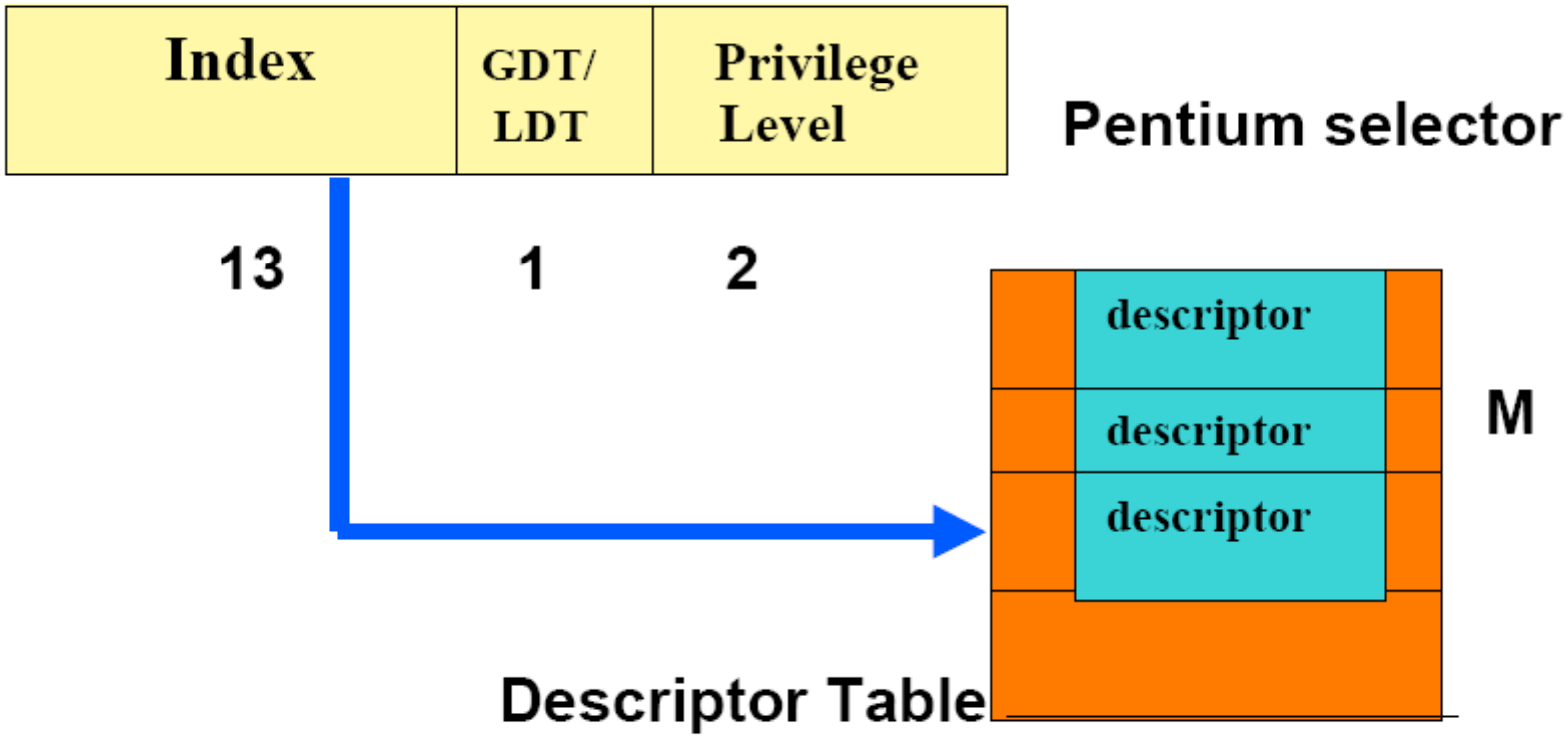
Intel Pentium Segmentation

- It allows a segment to be as large as 4GB, and the maximum number of segments per process is 16KB.

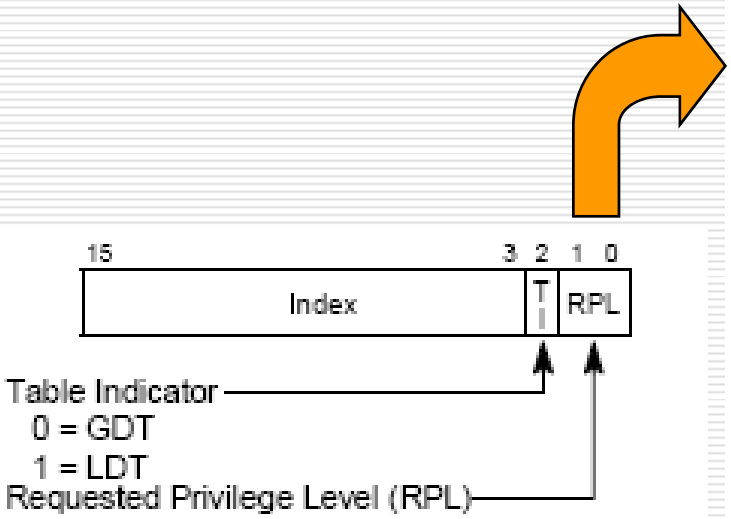


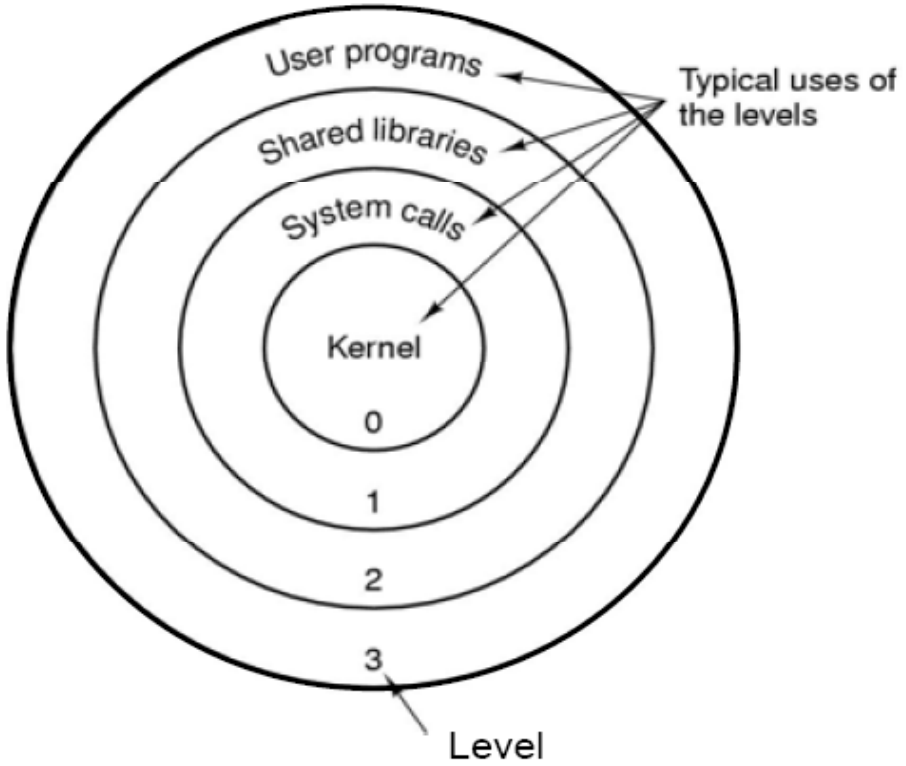
- Index designates the segment number
- G/L indicates whether the segment is in the GDT or LDT
  - GDT—Global Descriptor Table (one for the system)
  - LDT—Local Descriptor Table (one per process)
- RPL deals with protection

A segment register contains a 16-bit selector

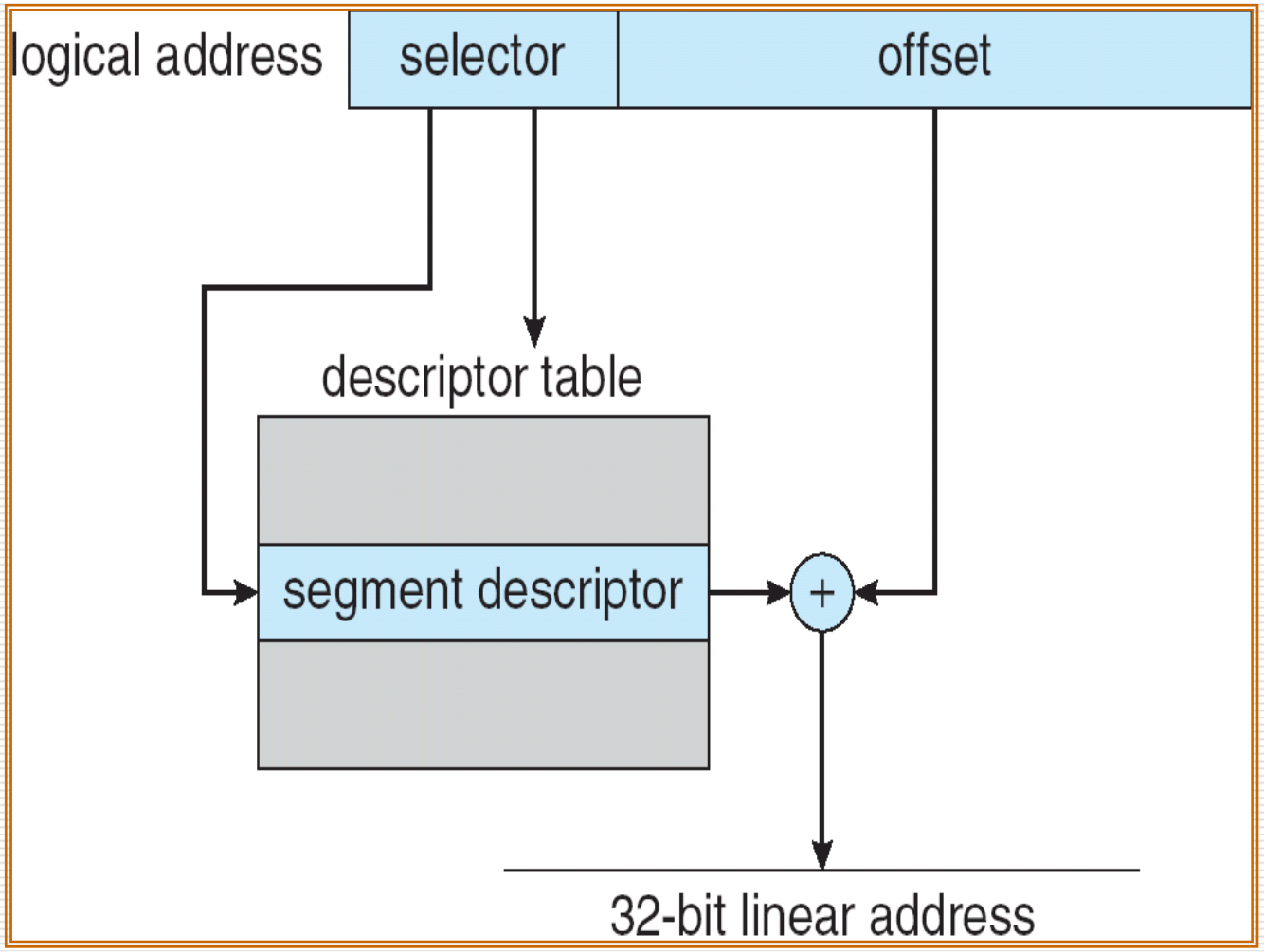


Intel Pentium Segmentation

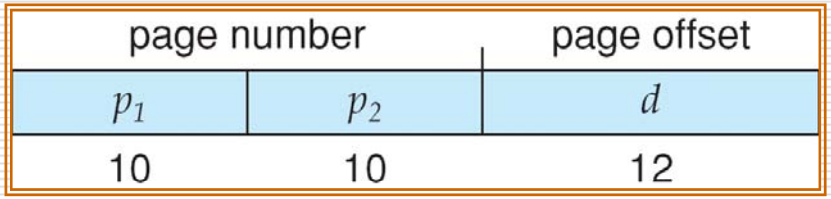


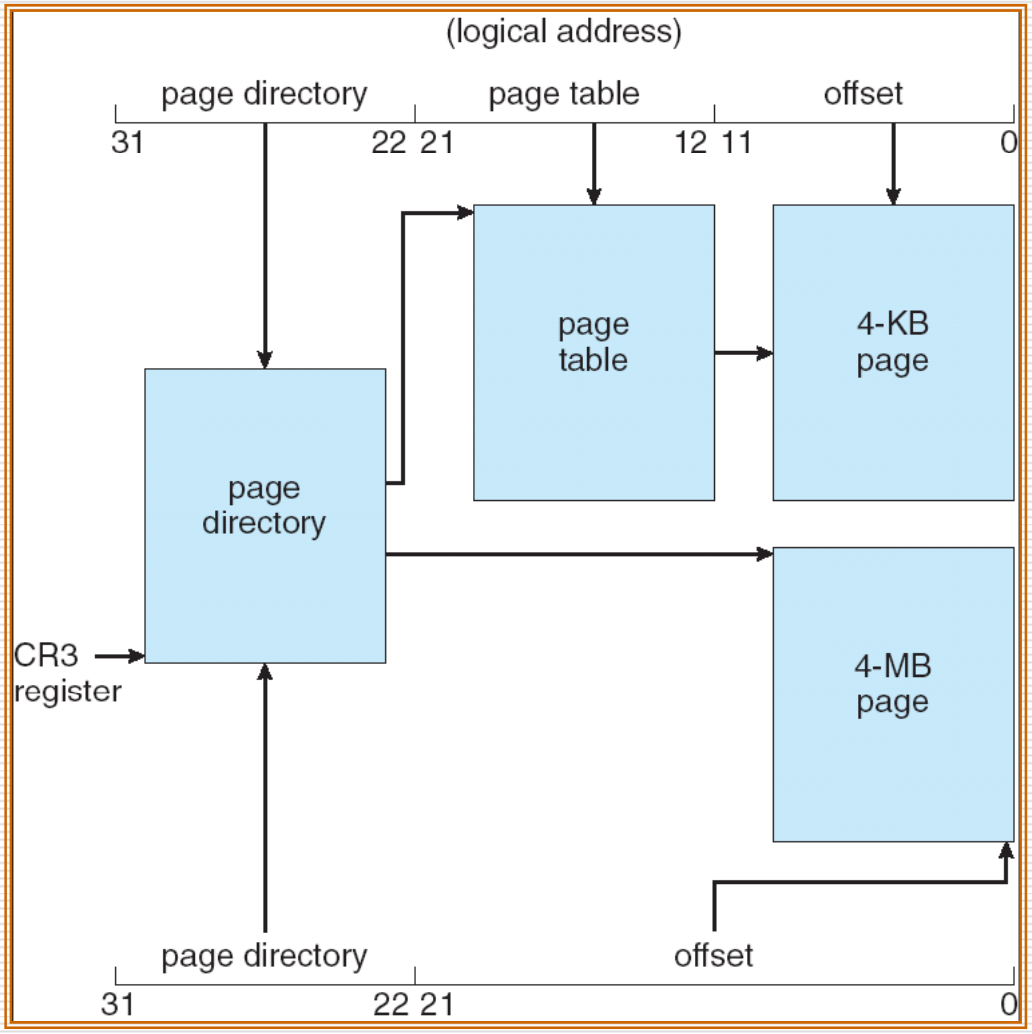


Intel Pentium Segmentation



Pentium Paging Architecture



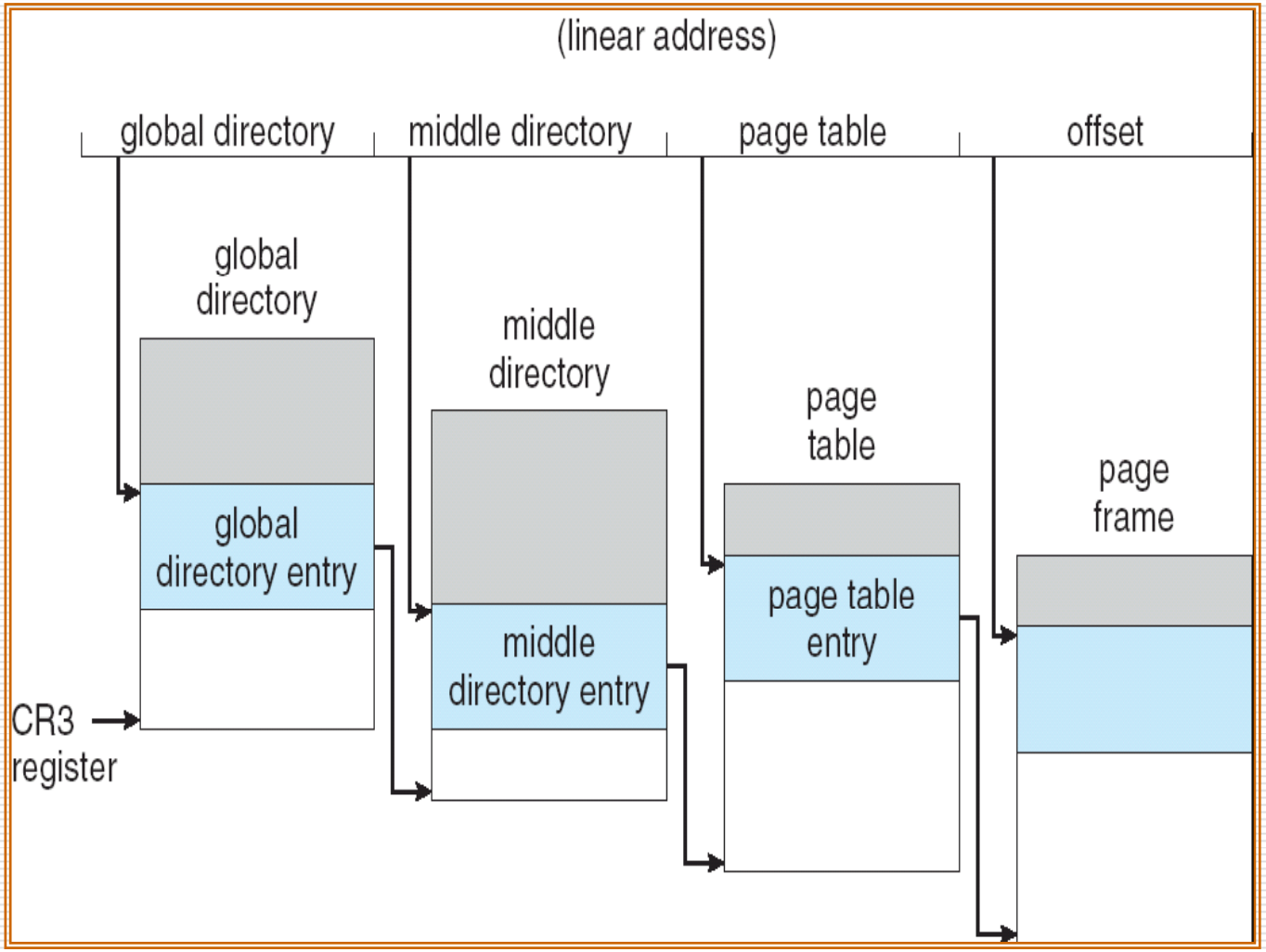


Three-level Paging in Linux

- Linux does rely on segmentation. On the Pentium, it uses only six segments.
  - A segment for kernel code
  - A segment for kernel data
  - A segment for user code
  - A segment for user data
  - A task-state segment
  - A default LDT segment

Linear Address in Linux

Broken into four parts:


(linear address)
global directory
middle directory
page table
offset
global directory
global directory entry
middle directory
middle directory entry
page table
page table entry
page frame
CR3 register

Assignments

- 8.5, 8.6, 8.9, 8.12

End of Chapter 8

Any Question?