

CYB 510 OPERATING SYSTEMS

Prepared by Prof. Dr. Lamiaa Fattouh

Lecture 8

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

CYB 510

MAIN MEMORY PART 2



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

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)**



Lecture 7

Associative Memory

- Associative memory – parallel search

Page #	Frame #

associative memory or
translation look-aside buffers
(TLBs)

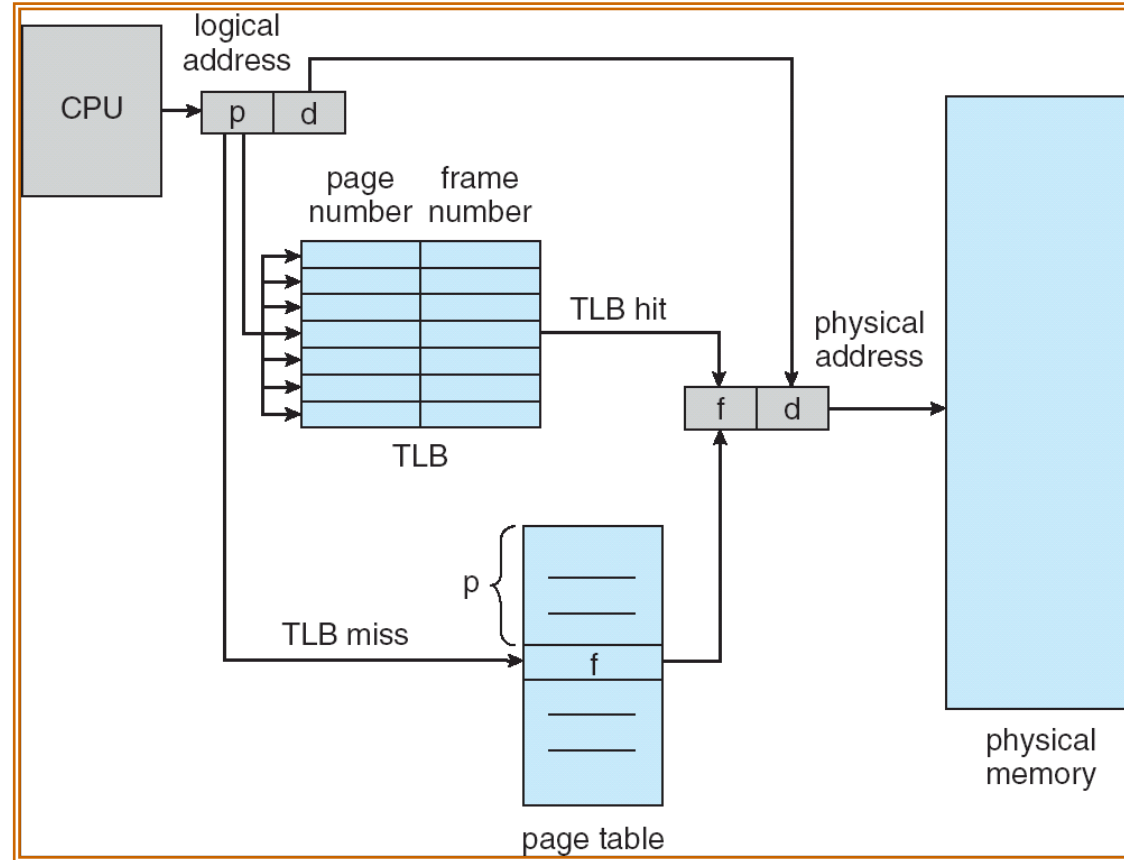
Address translation (A' , A'')

- If A' is in associative register, get frame # out
- Otherwise get frame # from page table in memory



Paging Hardware With TLB

associative memory or
translation look-aside buffers
(TLBs)



- 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; ration related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

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

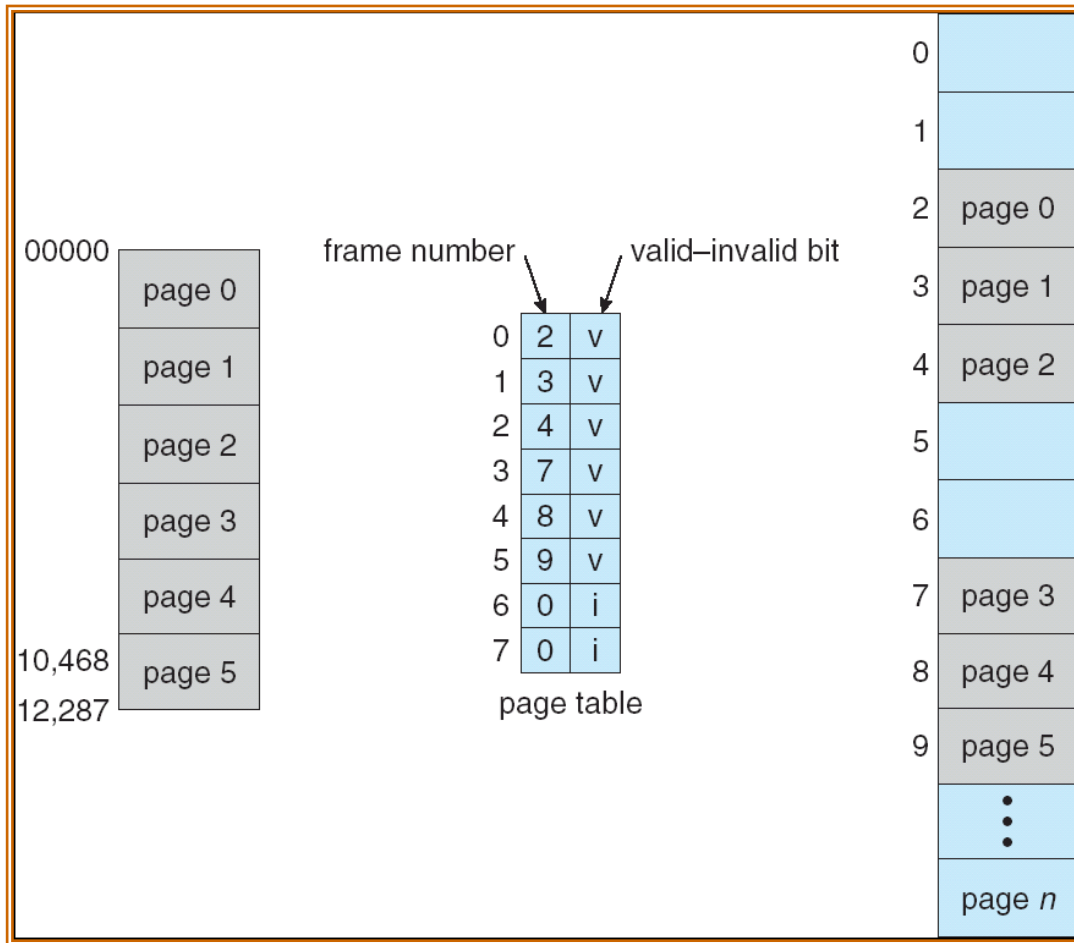


Memory Protection

- 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



- 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 Paging Example

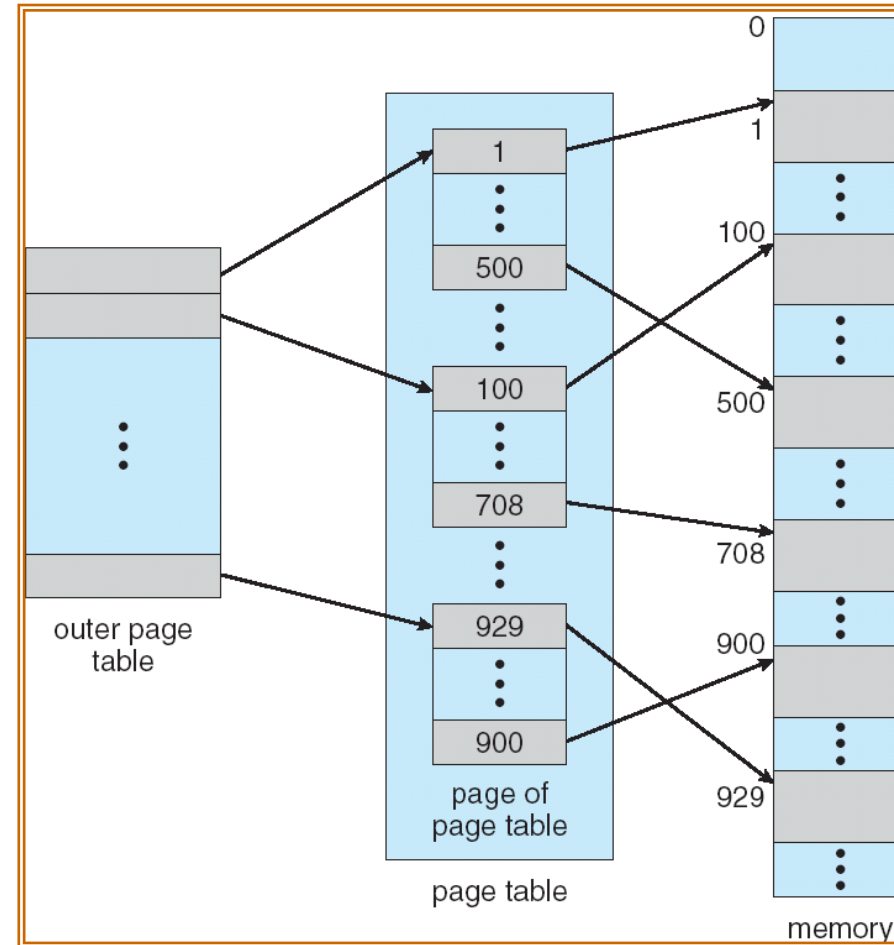
- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - a 10-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

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

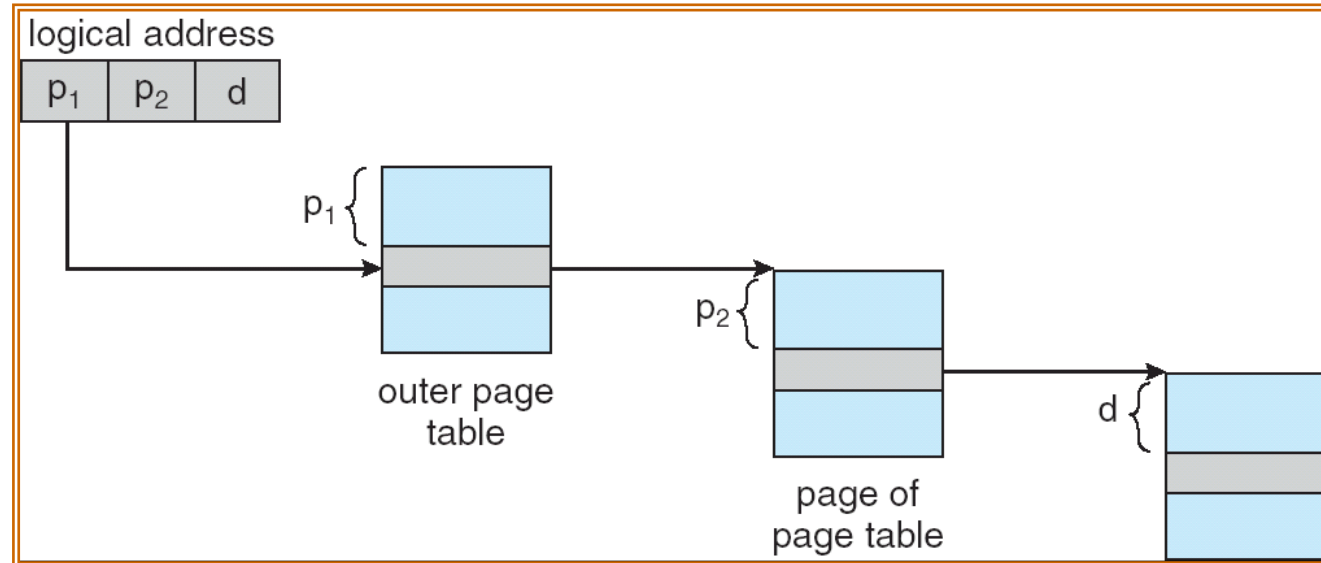


Two-Level Page-Table Scheme



Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

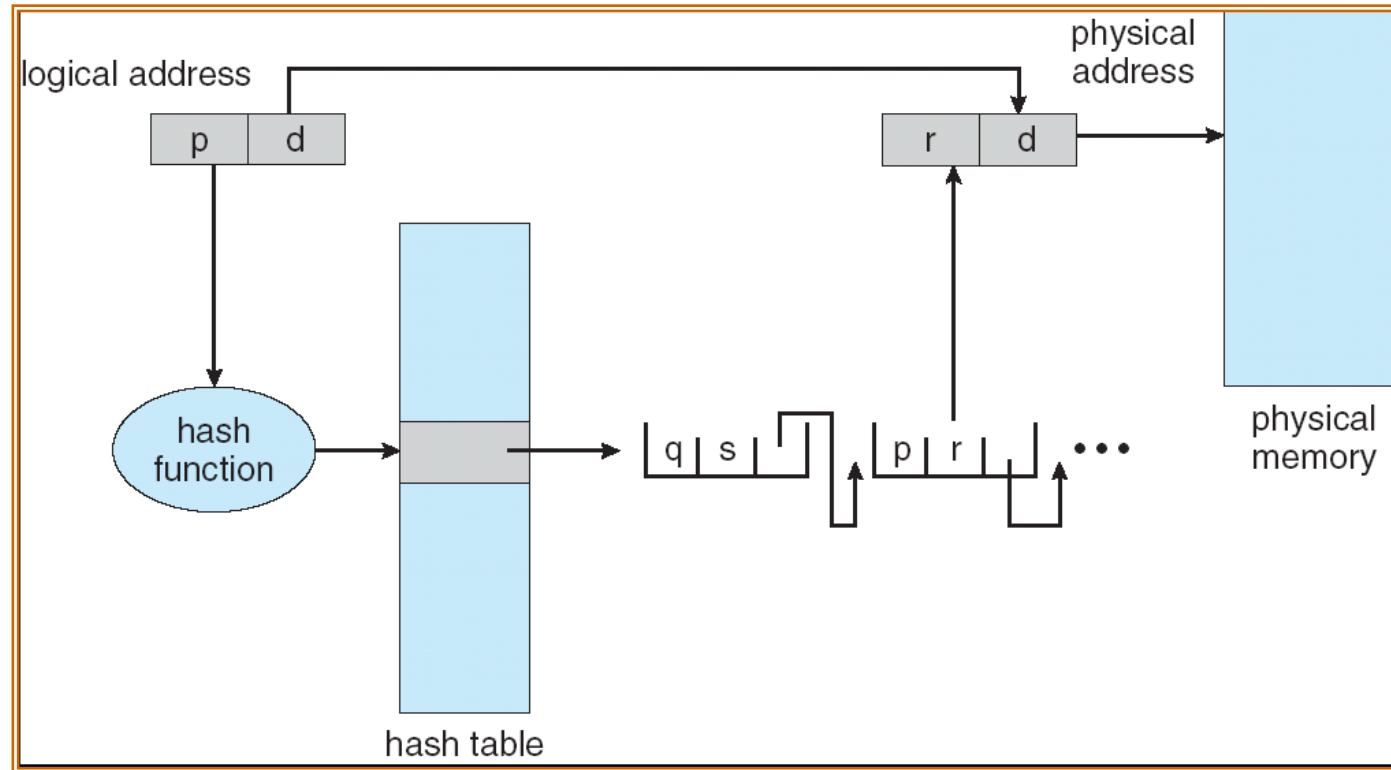


Hashed Page Tables

- 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

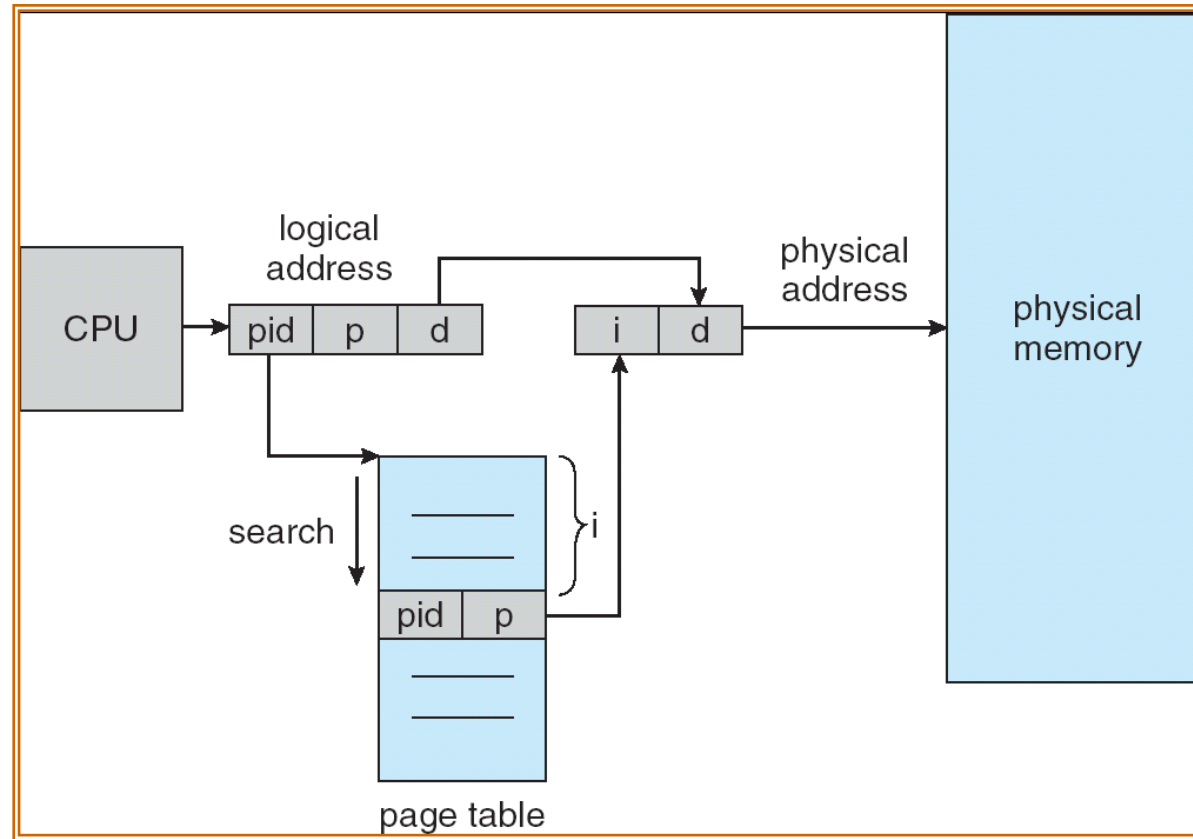


Inverted Page Table

- One entry for each real page of memory
- 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



Shared Pages

- **Shared code**

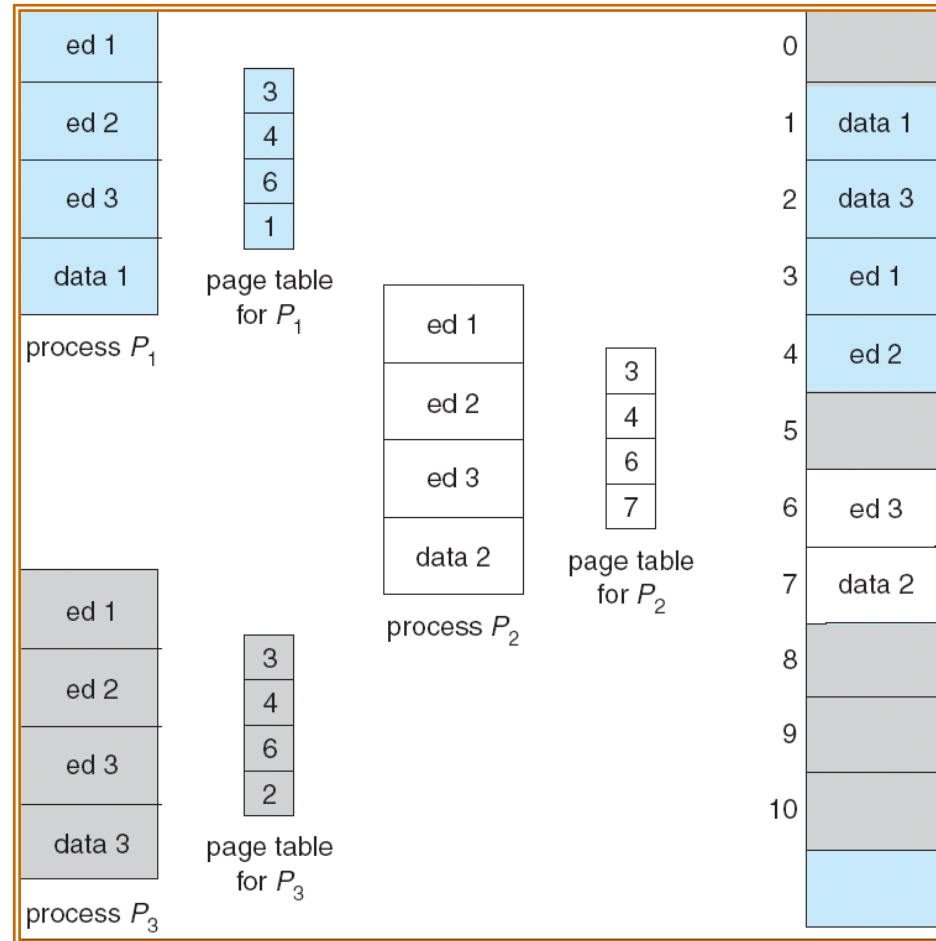
- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

- **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



Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of $2,048 - 1,086 = 962$ bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = $1 / 2$ frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
 - Solaris supports two page sizes – 8 KB and 4 MB



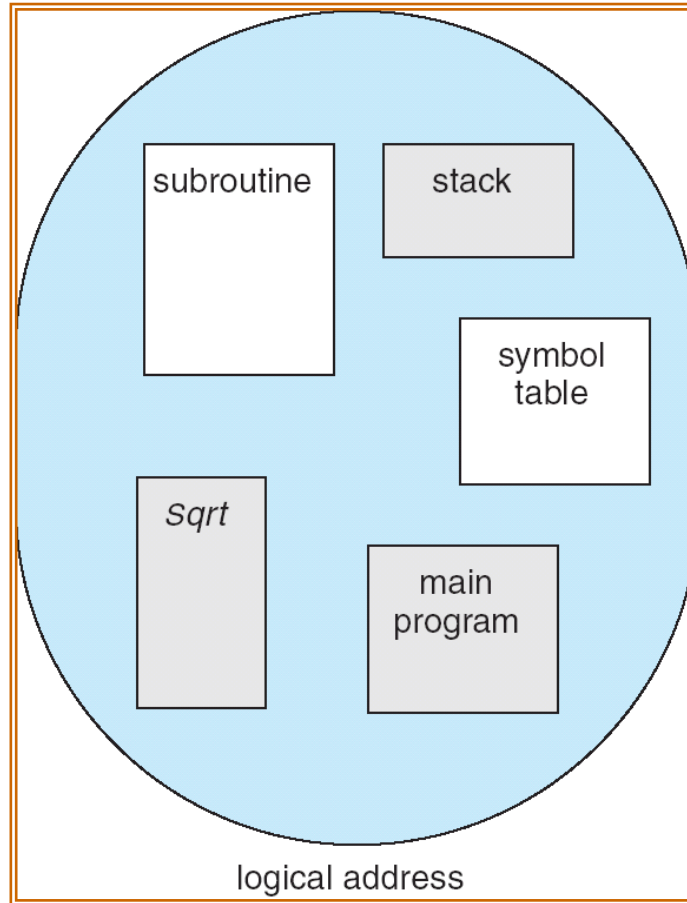
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

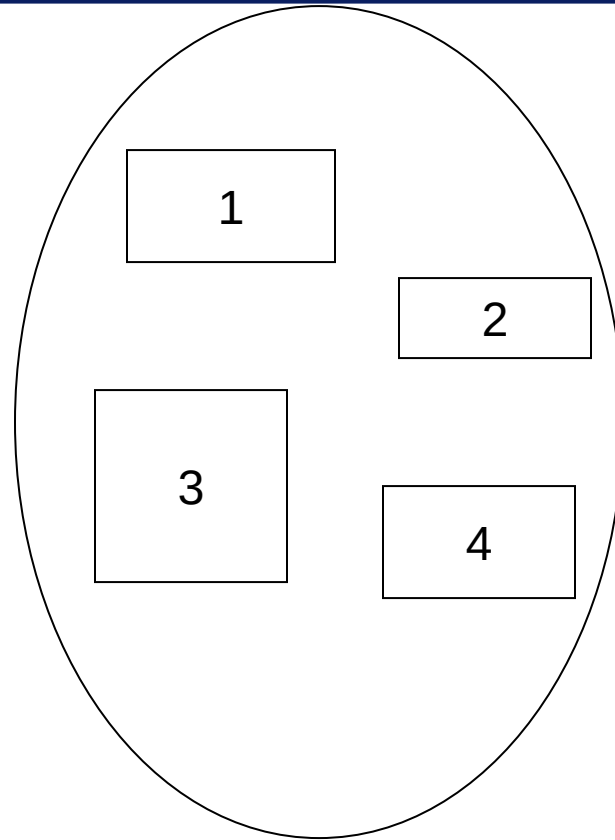
main program,
procedure,
function,
method,
object,
local variables, global variables,
common block,
stack,
symbol table, arrays



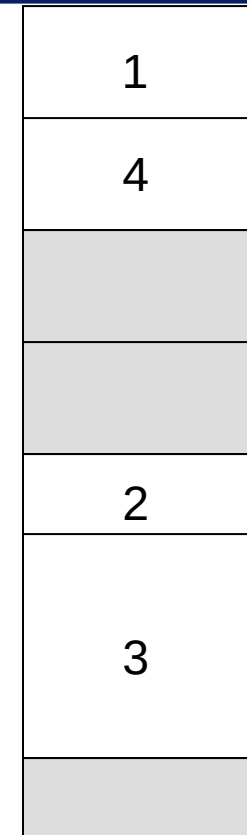
User's View of a Program



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 $s < \text{STLR}$



Segmentation Architecture (Cont.)

- **Sharing.**
 - shared segments
 - same segment number
- **Allocation.**
 - first fit/best fit
 - external fragmentation

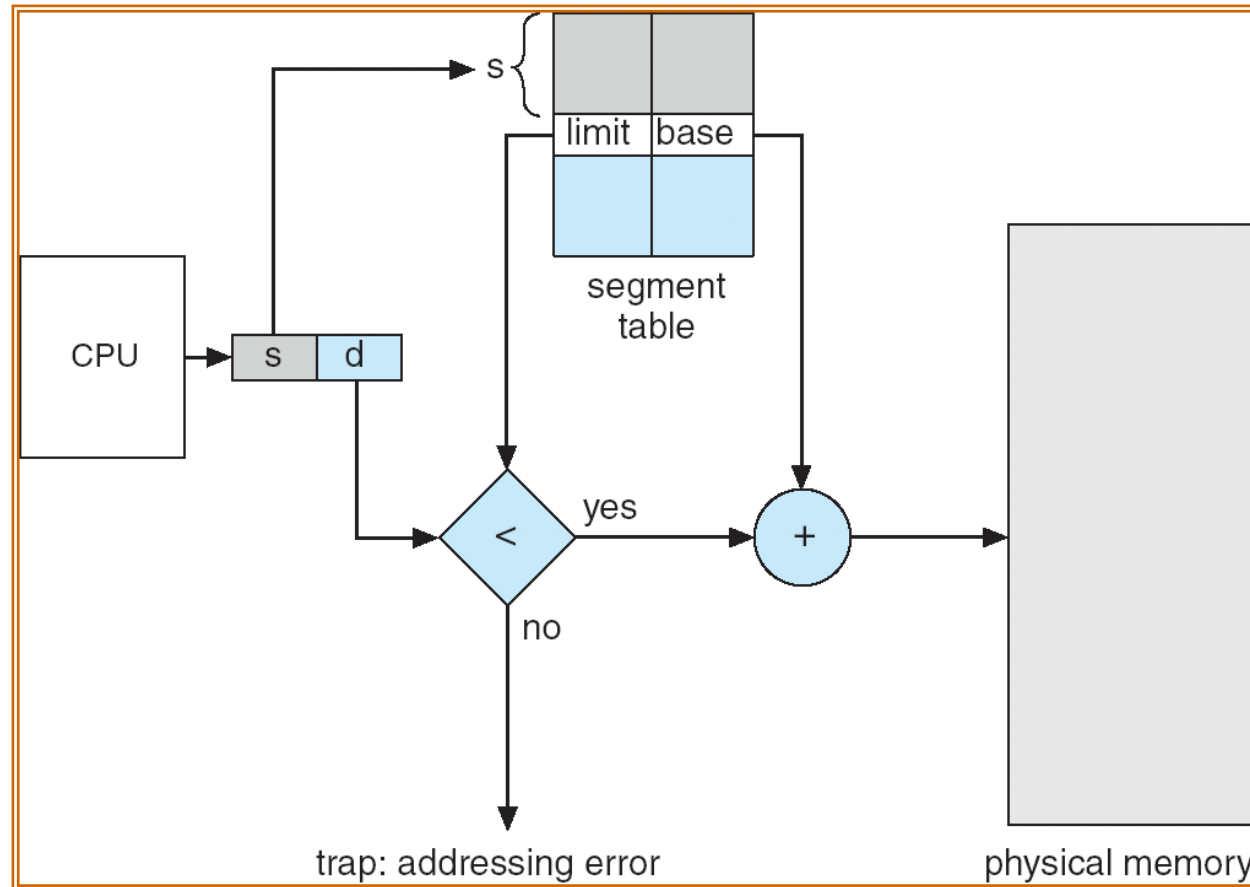


Segmentation Architecture (Cont.)

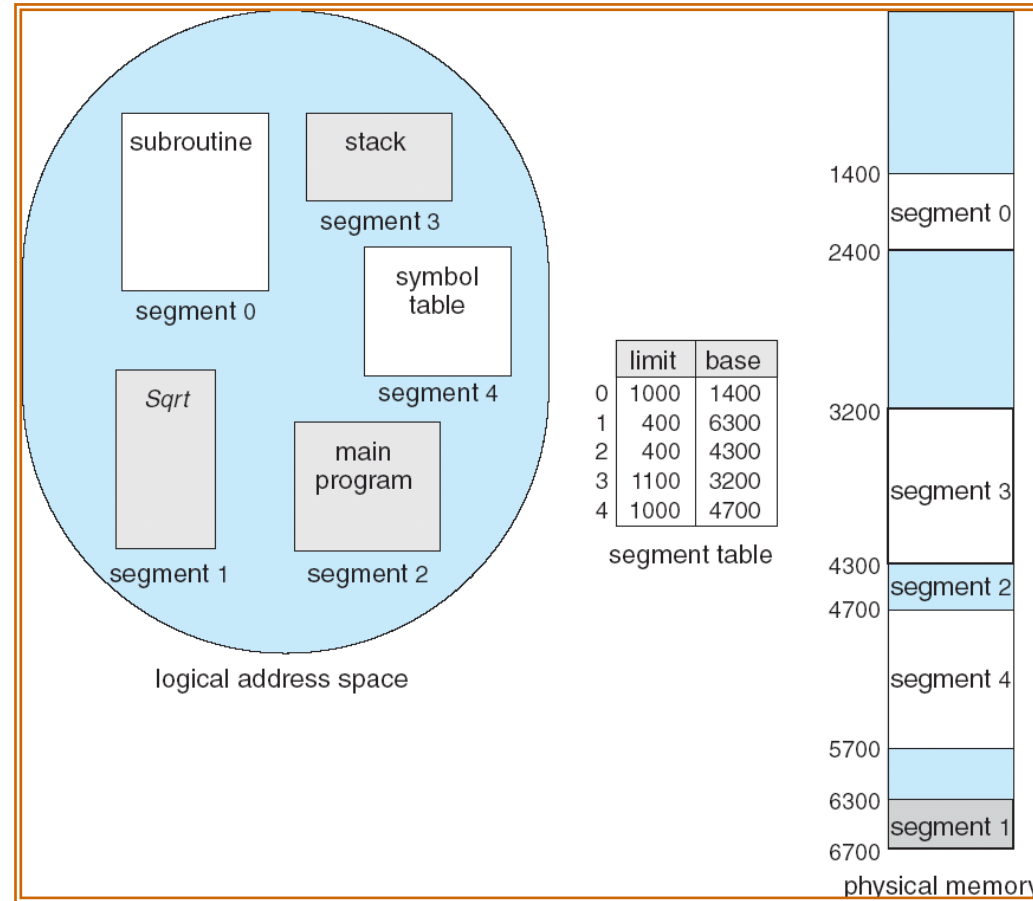
- Protection. With each entry in segment table associate:
 - validation bit = 0 $\Rightarrow$ illegal segment
 - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation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



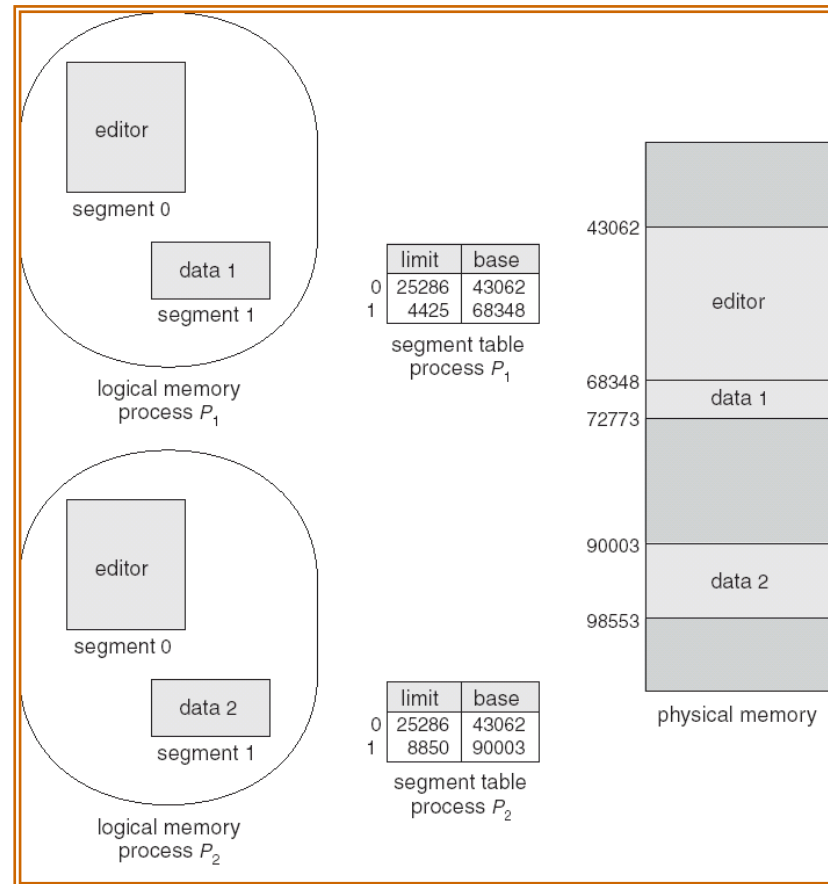
Address Translation Architecture



Example of Segmentation



Sharing of Segments

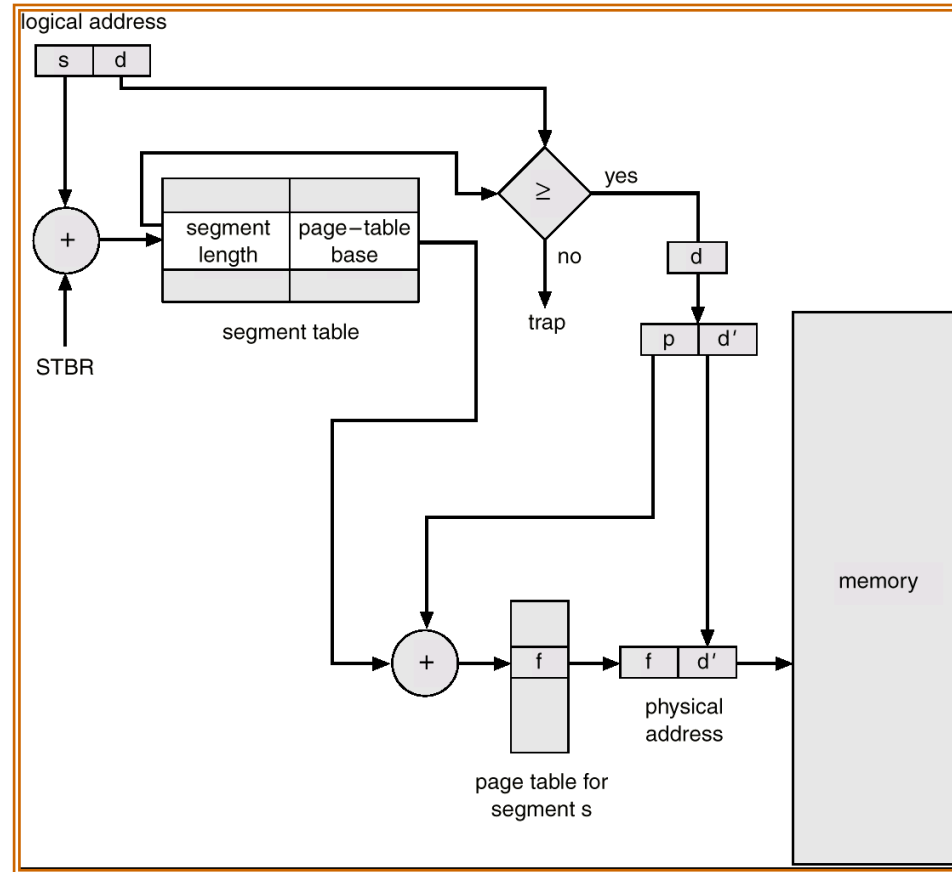


Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment



MULTICS Address Translation Scheme



Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips.
- Pentium CPUs are 32-bit and called IA-32 segmentation architecture.
- Current Intel CPUs are 64-bit and called IA-64 paging architecture.
- Notably, however, Intel's dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success.



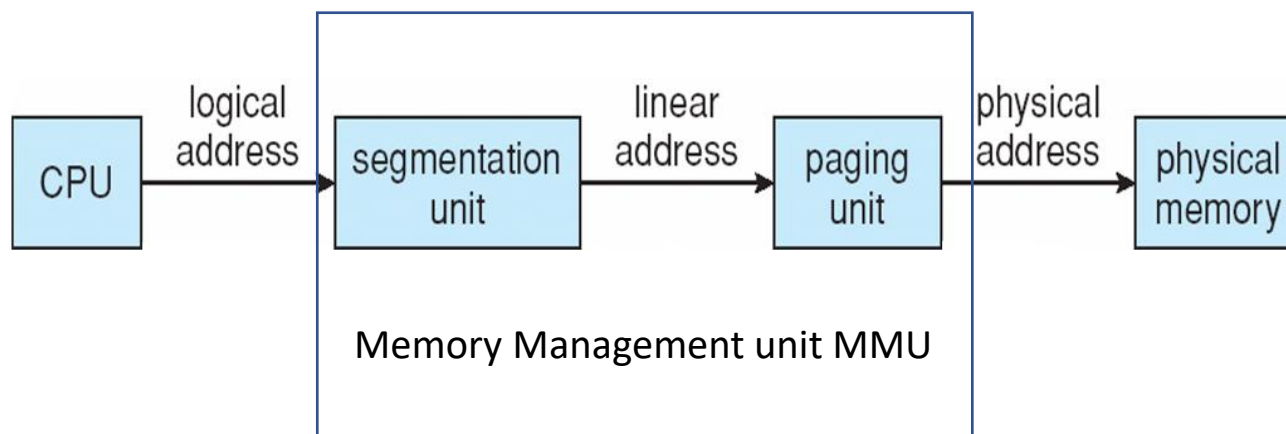
Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
 - Each segment can be as large as 4 GB
 - Up to 16 K segments per process
 - Pages sizes can be 4 KB or 4 MB



Example: The Intel IA-32 Architecture

Memory management in IA-32 systems is divided into two Components segmentation and paging



Linear address given to paging unit

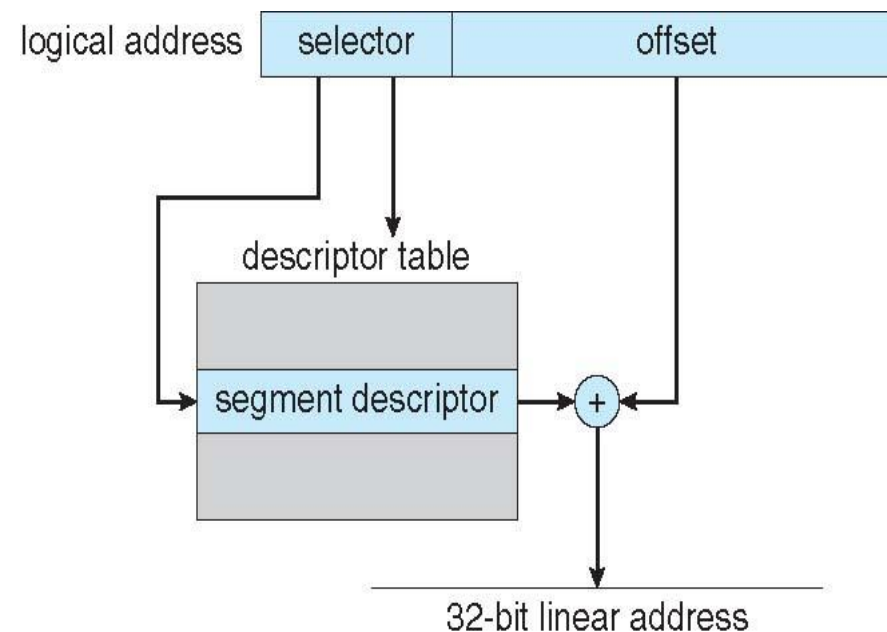
Which generates physical address in main memory

The Segmentation and Paging units form equivalent of MMU



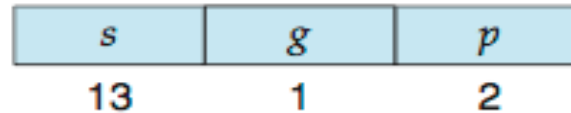
Example: The Intel IA-32 Architecture (Cont.)

- Logical address is Divided into two partitions
 - First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
 - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)
- Selector given to segmentation unit
 - Which produces linear addresses
 - Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment



Logical to Physical Address Translation in IA-32

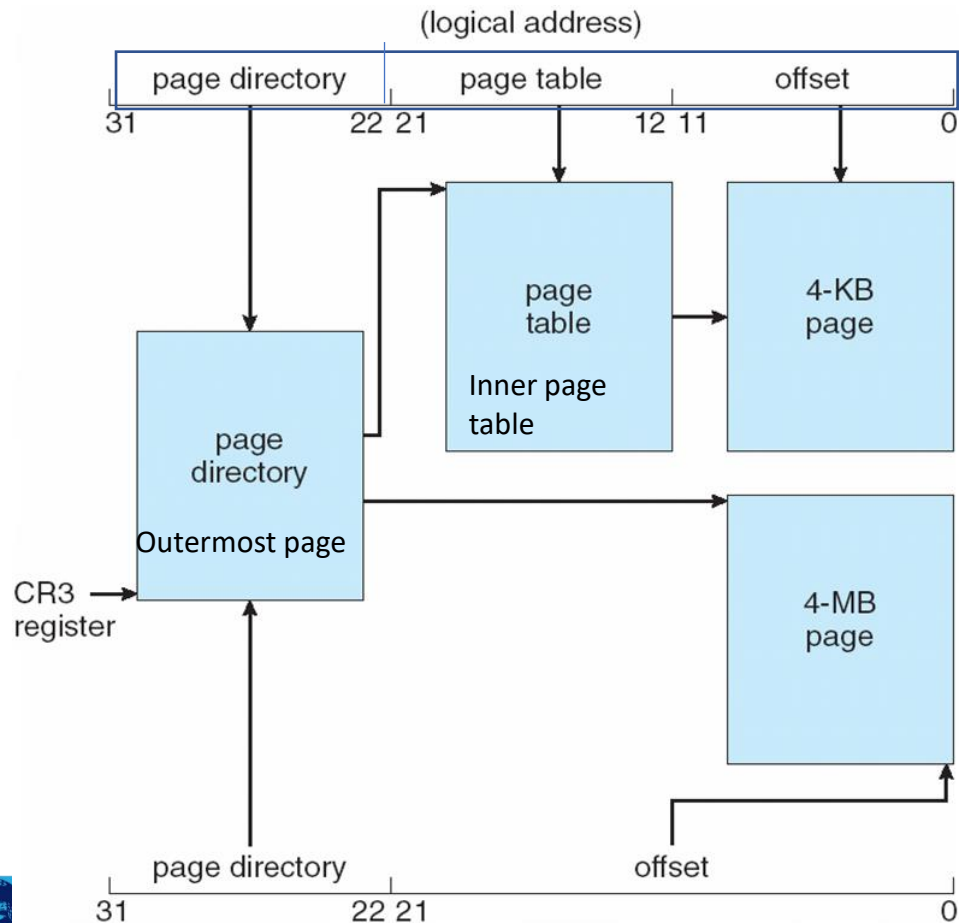
Selector is a 16-bit number



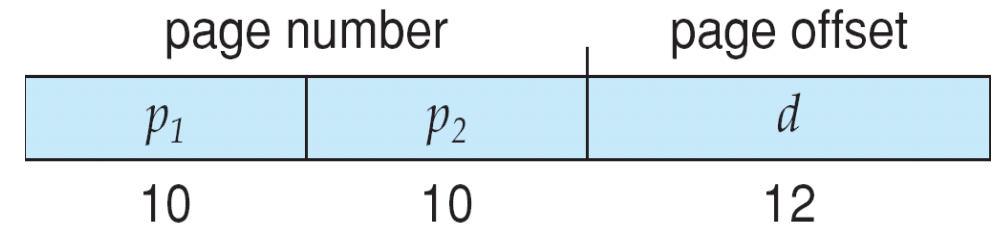
S designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.



Intel IA-32 Paging Architecture



32-bit linear address



The CR3 register points to the page directory for the current process.

One entry in the page directory is the Page Size flag, which—if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB.

If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

Problem 1

- Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?



Solution

- Let p1, p2, p3 & p4 are the names of the processes

- First-fit:

- **P1>>>** 100, 500, 200, 300, 600

212 KB, 417 KB, 112 KB, and 426 KB

- **P2>>>** 100, 288, 200, 300, 600

- **P3>>>** 100, 288, 200, 300, 183

- 100, 176, 200, 300, 183 <<<<< **final set of hole**

- P4 (426K) must wait



Solution

- Best-fit:

- **P1>>>** 100, 500, 200, **300**, 600

- **P2>>>** 100, **500**, 200, 88, 600

212 KB, 417 KB, 112 KB, and 426 KB

- **P3>>>** 100, 83, **200**, 88, 600

- **P4>>>** 100, 83, 88, 88, **600**

- 100, 83, 88, 88, 174 <<<<< **final set of hole**



- Worst-fit:

- **P1>>>** 100, 500, 200, 300, 600
- **P2>>>** 100, 500, 200, 300, 388
- **P3>>>** 100, 83, 200, 300, 388
- 100, 83, 200, 300, 276 <<<<< **final set of hole**
- P4 (426K) must wait

212 KB, 417 KB, 112 KB, and 426 KB



Solution

- In this example, Best-fit turns out to be the best ***because there is no wait processes.***



Problem 2

Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

- a. 3085
- b. 42095

a. $3085 \rightarrow 3085 - (3 * 1024) = 3085 - 3072 = 13$
page number 3 offset 13



Problem 2

b) $42095 \rightarrow 42095/1024 = 41.1$

$$41 * 1024 = 41984$$

$$42095 - 41984 = 111$$

Page number = 41 and offset = 111



Problem 3

Consider the following segment table:

<u>Segment</u>	<u>Base</u>	<u>Length</u>
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

- a. 0,430
- b. 1,10
- c. 2,500

Answer:

- a. $219 + 430 = 649$
- b. $2300 + 10 = 2310$
- c. illegal reference, trap to operating system



Summary

- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.
- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.



Summary

- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.



Summary

- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.



Summary

- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.



- ❑ Various techniques
 - ❑ **Fixed partitions** – easy to use, but internal fragmentation
 - ❑ **Variable partitions** – more efficient, but external fragmentation
 - ❑ **Paging** – use small, fixed size chunks, efficient for OS
 - ❑ **Segmentation** – manage in chunks from user's perspective
 - ❑ **Combine paging and segmentation** to get benefits of both



END LECTURE 8

