Page Table Implementation

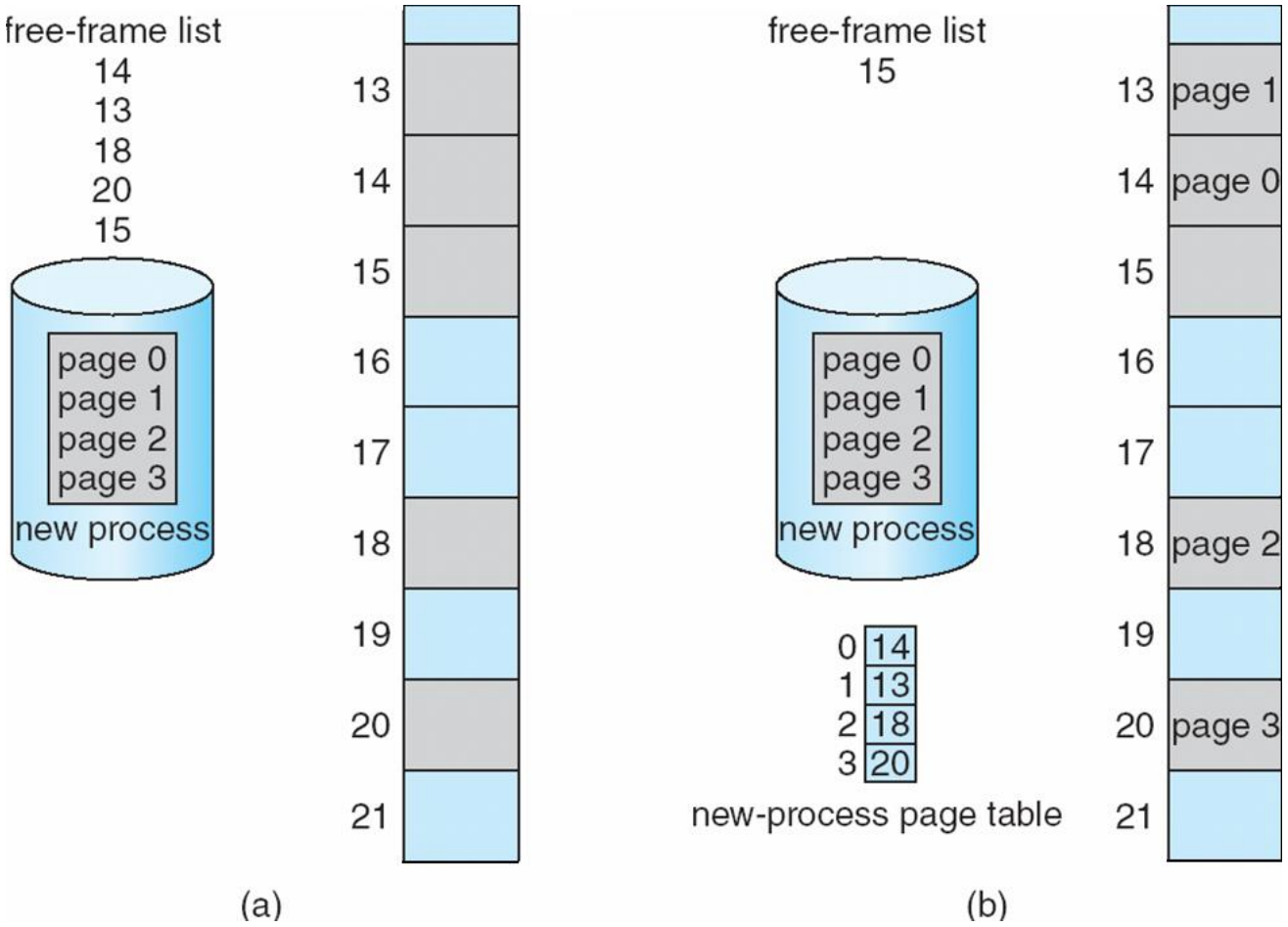
Outline

- Structure of the Page Table
- Implementation Issues
- Segmentation

Frames

- Each process page needs one frame.
- The OS needs to be aware of the available frames.
- This is through the frame table
  - Each frame has an entry in the table
  - Each entry determines if the frame is allocated or not
    - If allocated indicates which page

Free Frames



Before allocation

After allocation

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 (PTLR) indicates size of the page table

Implementation of Page Table

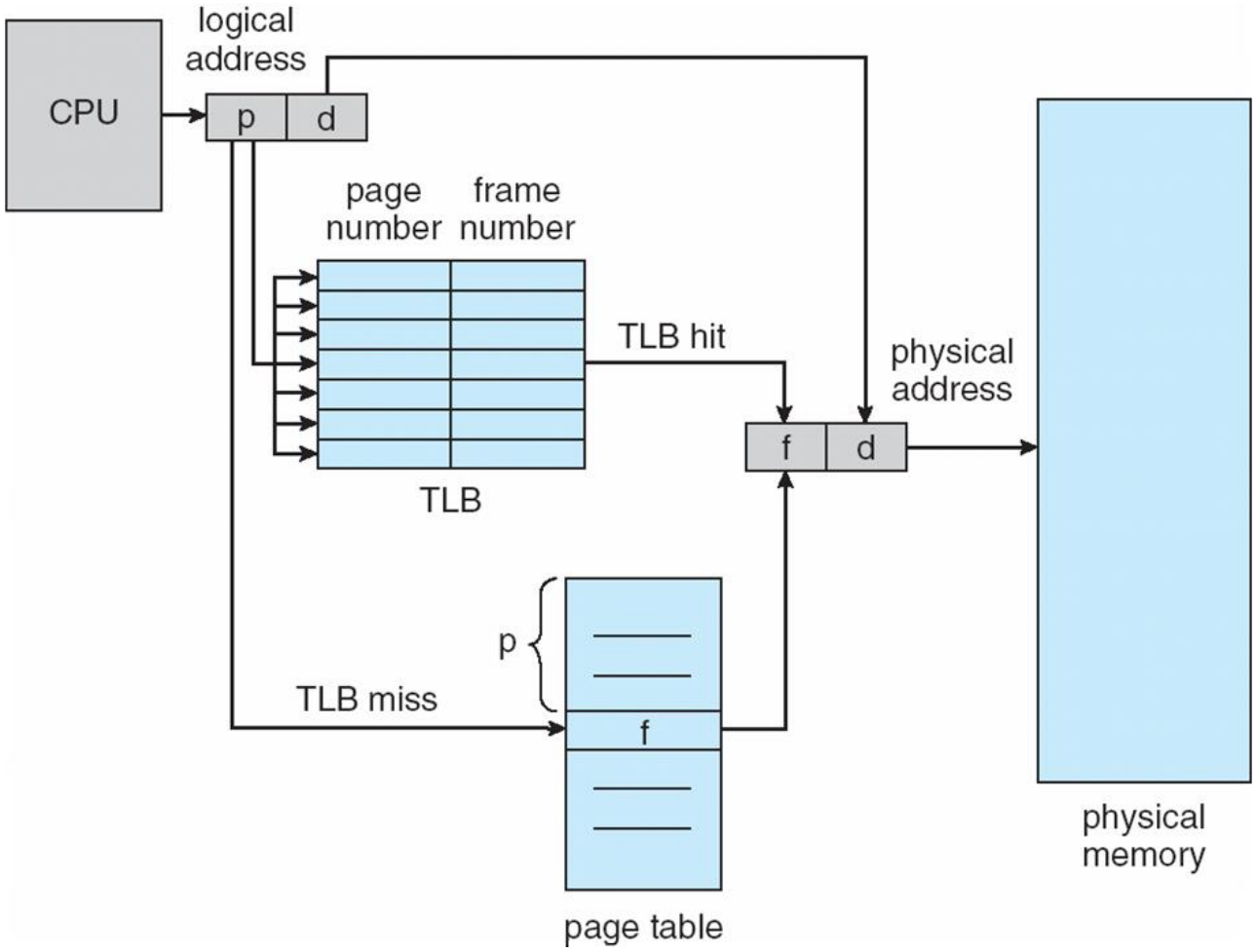
- Problem: Every data/instruction access requires two memory accesses:
  - Access for the page table
  - Access for the data/instruction.
- Solution: Use a special fast-lookup hardware cache called associative memory or Translation Look-aside Buffers (TLBs)
- TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

TLB

- Associative memory – parallel search
- Address translation (p, d)
  - If p is in associative register, get frame number out
  - Otherwise get frame number from page table in memory

- Address-space identifiers (ASIDs) are associated with each entry
  - uniquely identifies each process to provide address-space protection for that process

Paging Hardware with TLB



Page Table with TLB (cont.)

- TLB: Cache for VM address to PM address lookup.
  - Like a dedicated cache for lookup.
  - TLB entries contain both virtual address and its corresponding physical address.
- Page Table: Map VM address to PM address as well, but it is slow.
- hardware cache called associative memory or Translation Look-aside Buffers (TLBs)
- TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

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)
- Example 1: 80% hits, 100 nanosecs to access memory and 20 nanosecs to search TLB. Determine EAT?
- Example 2: 98% hits, determine EAT?

Effective Access Time (cont.)

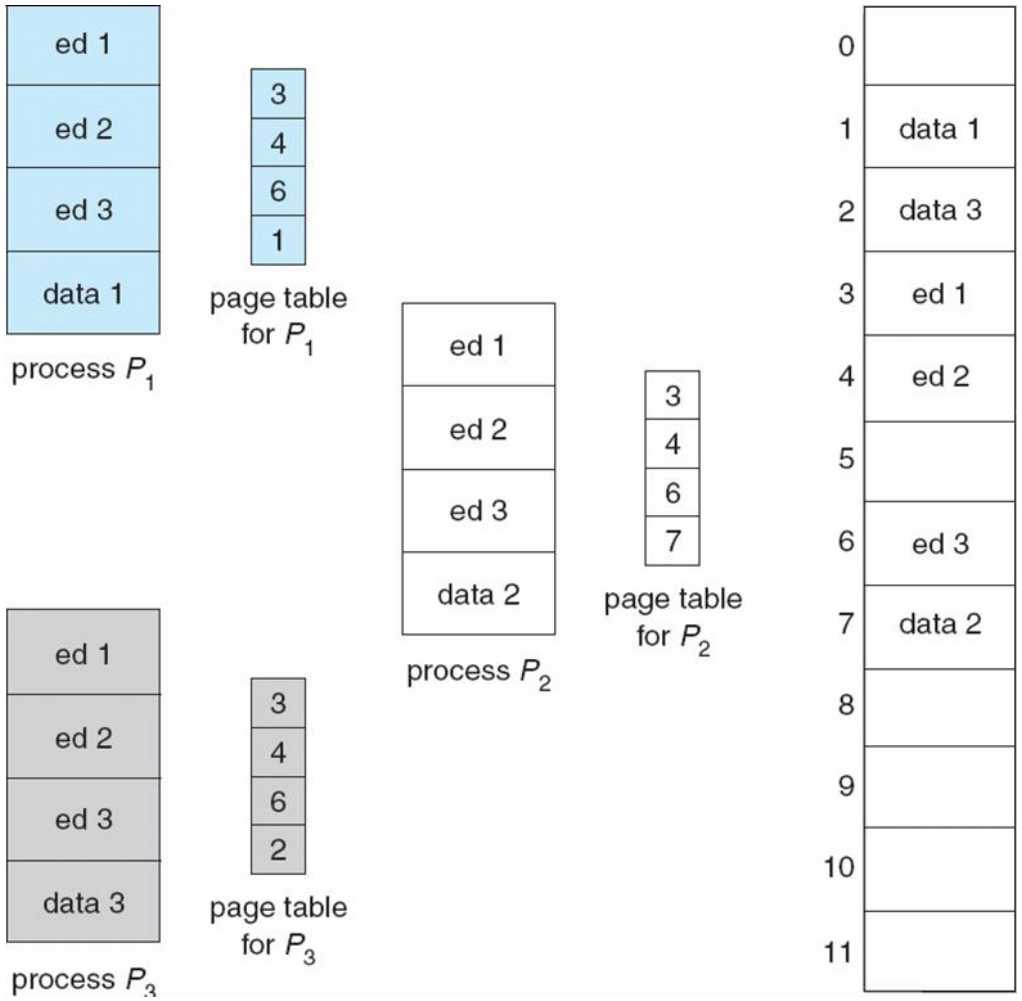
- TLB hit time = TLB search time + memory access time
- TLB miss time = TLB search time + memory access time + memory access time + memory access time.
- EAT = TLB miss time * (1- hit ratio) +
  TLB hit time * hit ratio
- Example 1: EAT = 0.80 * 120 + 0.20 * 220 = 140ns
- Example 2: EAT = 0.98 * 120 + 0.02 *220 = 122ns

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



Structure of the Page Table

- Typically, systems have large logical address spaces (2^{32} to 2^{64})
- Page table could be excessively large
- Example:
  - 32-bit logical address space
    - The number of possible addresses in the logical address space is 2^{32}
  - Page size is 4 KB (2^{12})
  - Page table may consist of up to 1 million entries ($2^{32} / 2^{12}$)
  - Assume each entry consists of 4 bytes
  - A process may need up to 4 MB for the page table

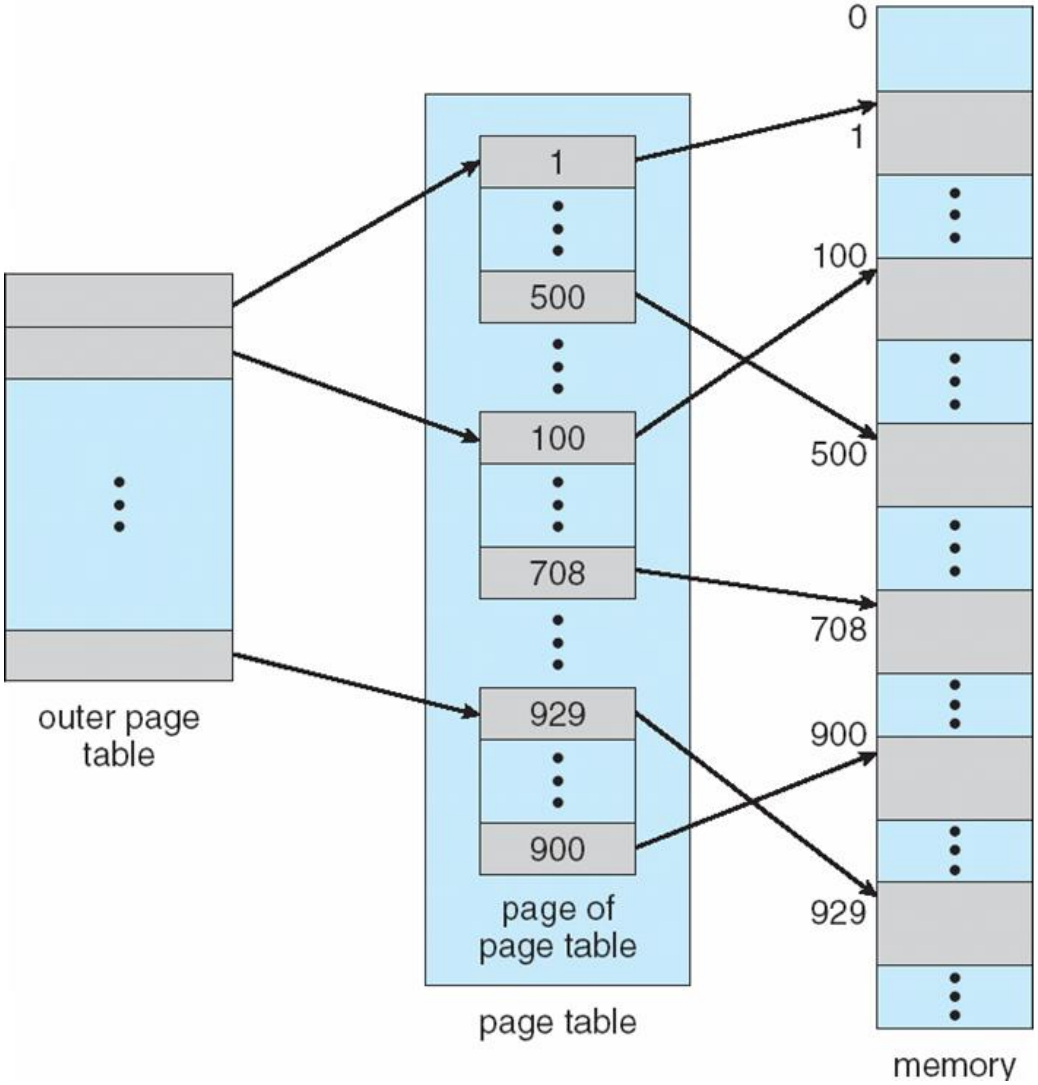
Structure of the Page Table

- The mapping should be fast
- Note that each memory reference requires access to the page table
- A typical instruction has an instruction word, and often a memory operand
- There is the potential for multiple page table references per instruction
- Why not have a single page table consisting of an array of fast hardware registers
  - Expensive in terms of registers used

Structure of Page Table

- Several approaches
  - Hierarchical Paging (multiple levels)
  - Hashed Page Tables
  - Inverted Page Tables

Two-Level Page-Table Scheme

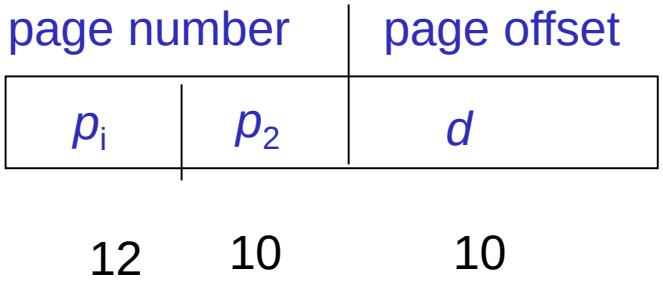


Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - A page number consisting of 22 bits
  - A page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - A 12-bit page number
  - A 10-bit page offset

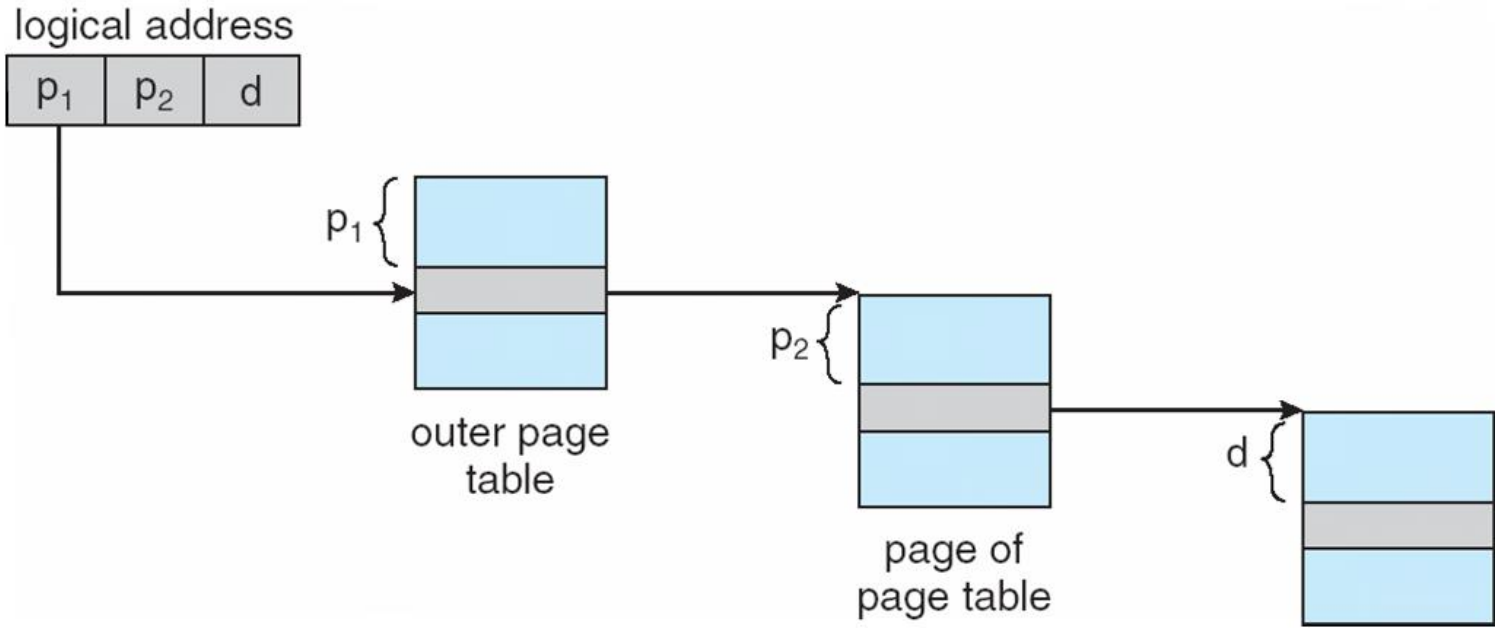
Two-Level Paging Example(cont.)

- Thus, a logical address is as follows:



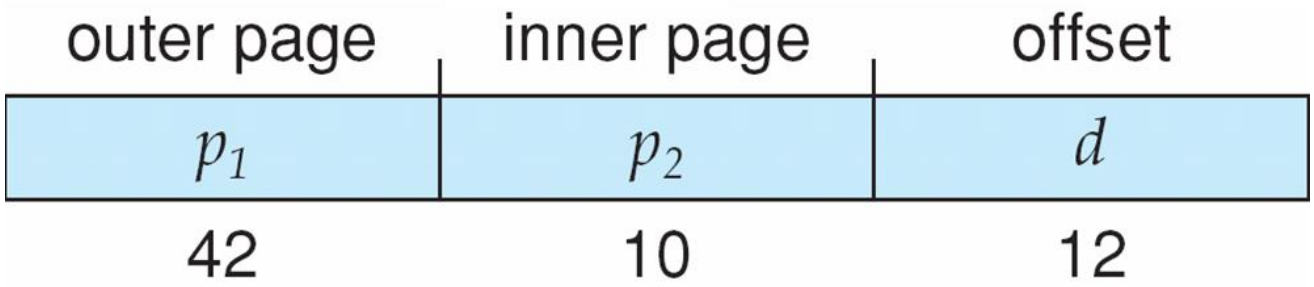
- Here pi is an index into the outer page table, and p2 is the displacement within the page of the outer page table

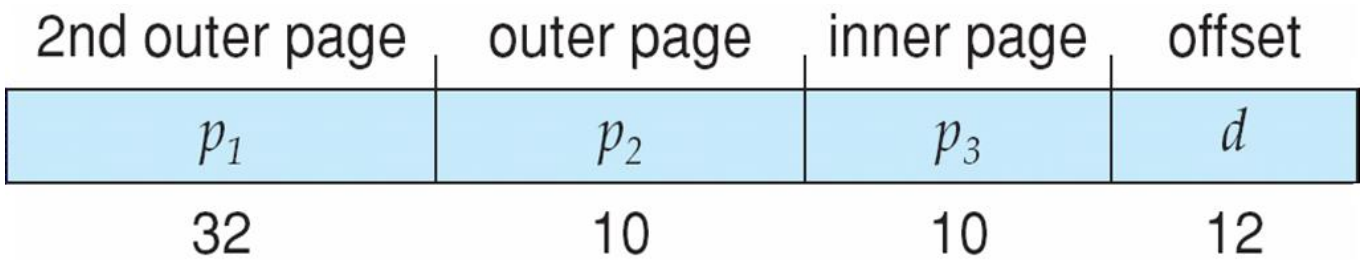
Address-Translation Scheme



- p_1 is used to index the outer page table
- p_2 is used to index the page table

Three-level Paging Scheme





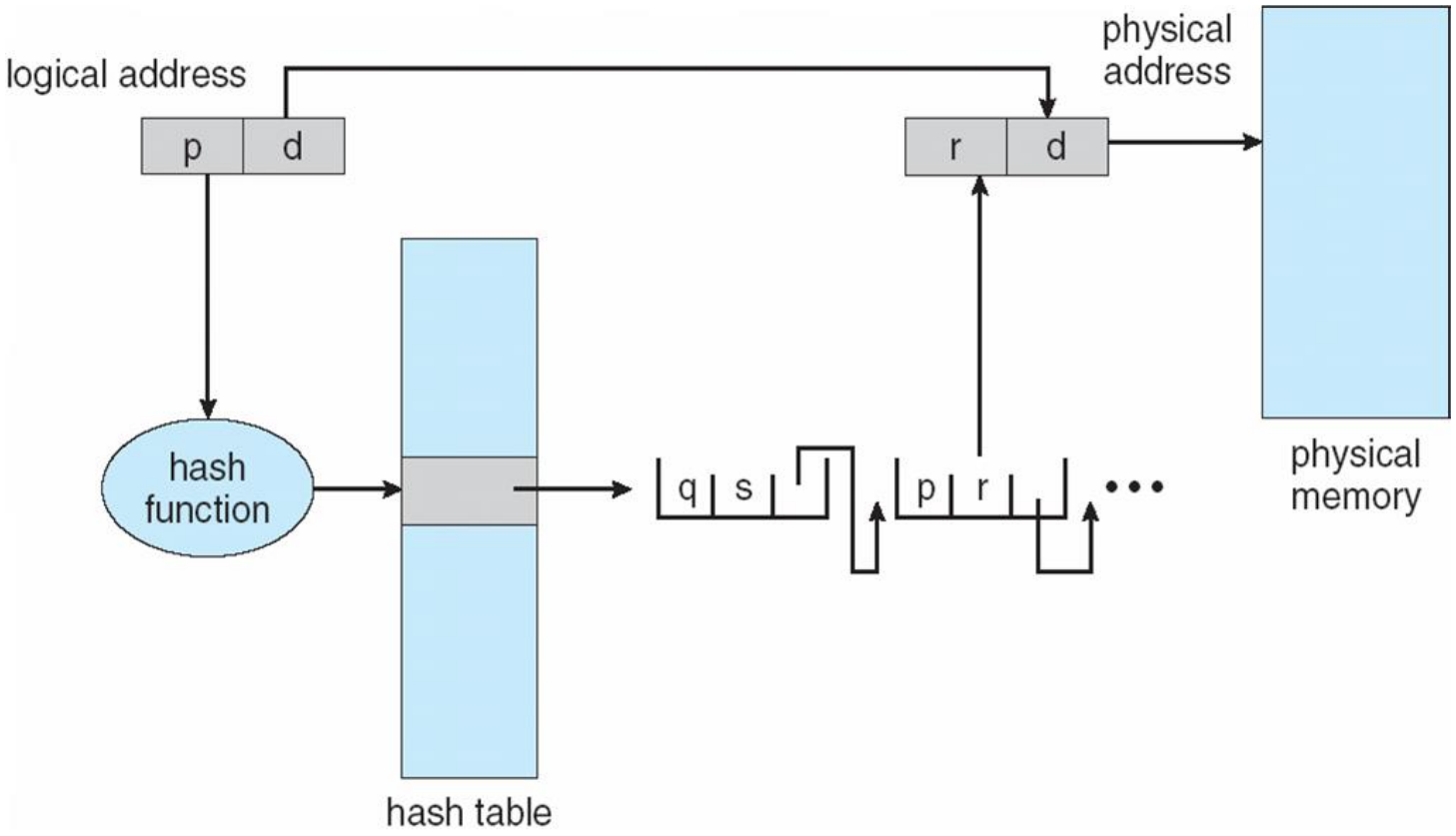
Hierarchical Paging

- What if you have a 64-bit architecture?
- Do you think hierarchical paging is a good idea?
  - How many levels of paging are needed?
  - What is the relationship between paging and memory accesses?

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
- To address this problem, use a hash table
- Used in the 64-bit UltraSparc and PowerPC

Inverted Page Table Architecture

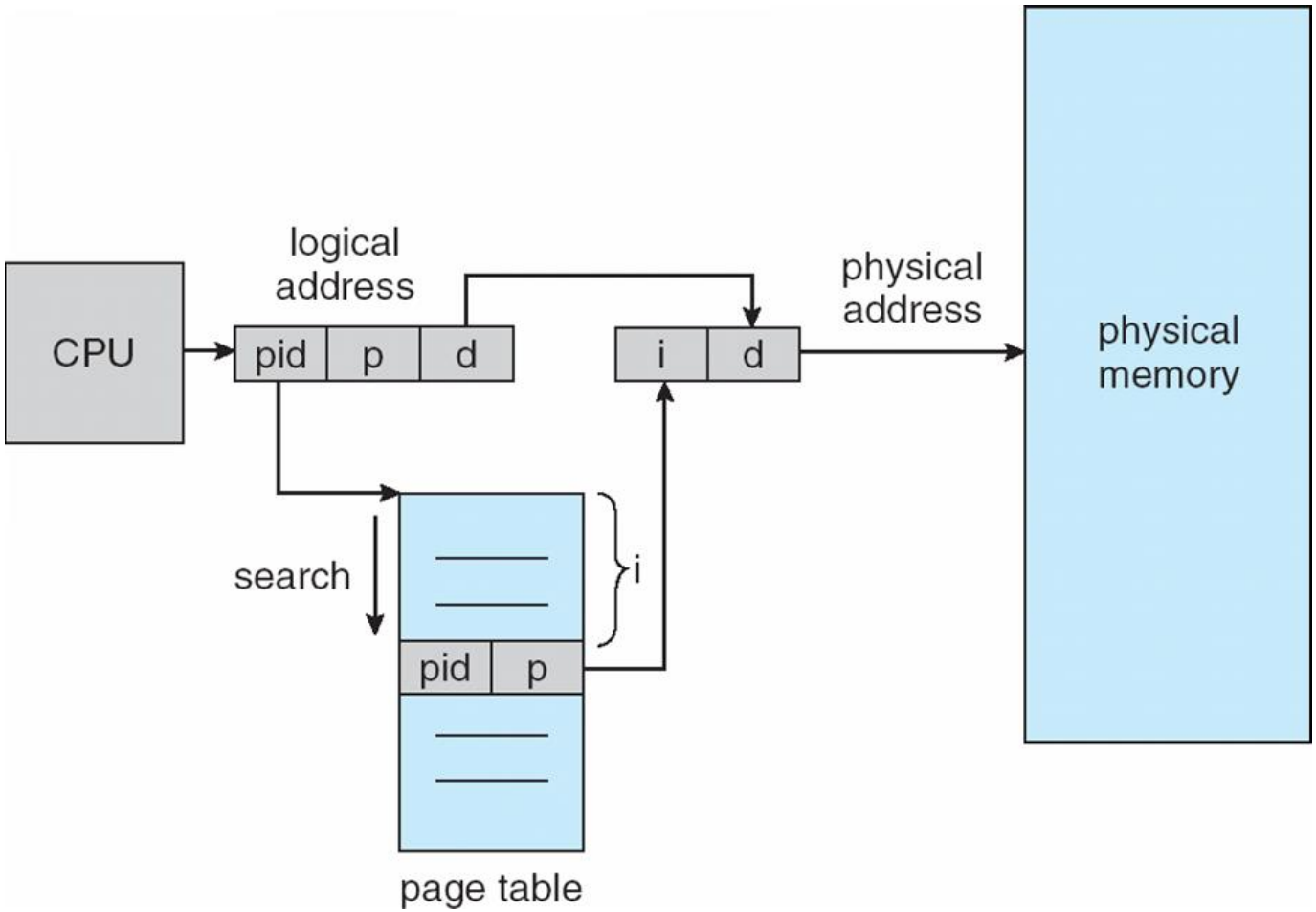
Logical address (p1)
Page no | offset

Page 0, process 1

Inverted Page Table

0		
1	Page0	p3
2	page0	p2
3	Page0	p1
4		
5	page3	p1
6	page4	p3

Inverted Page Table Architecture (cont.)

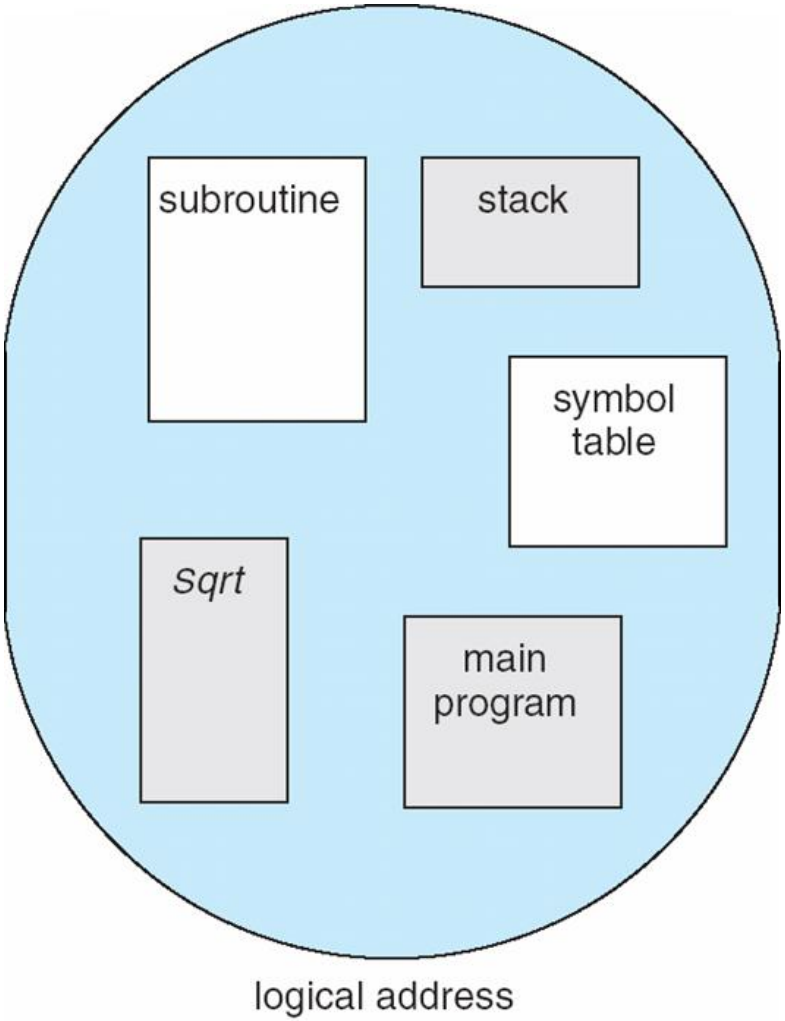


Segmentation

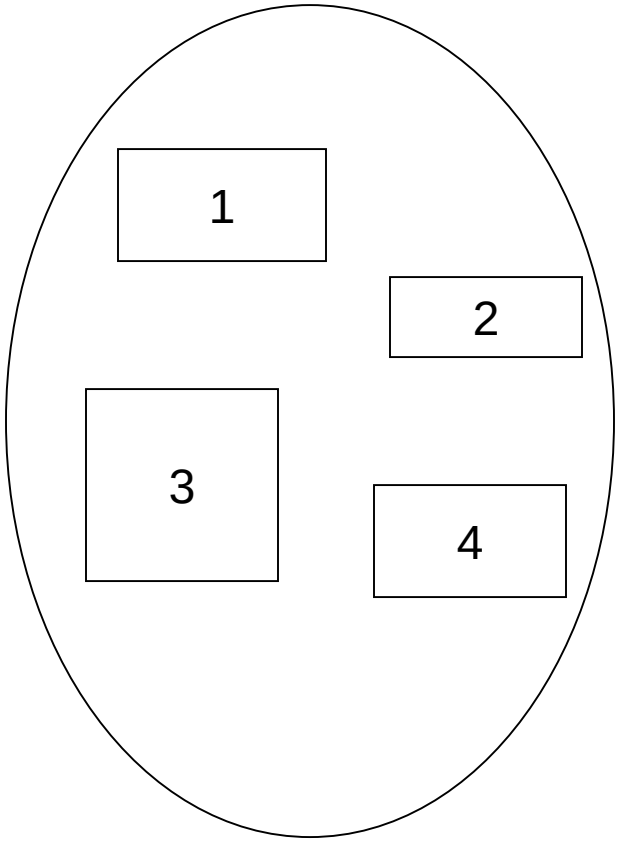
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    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>,

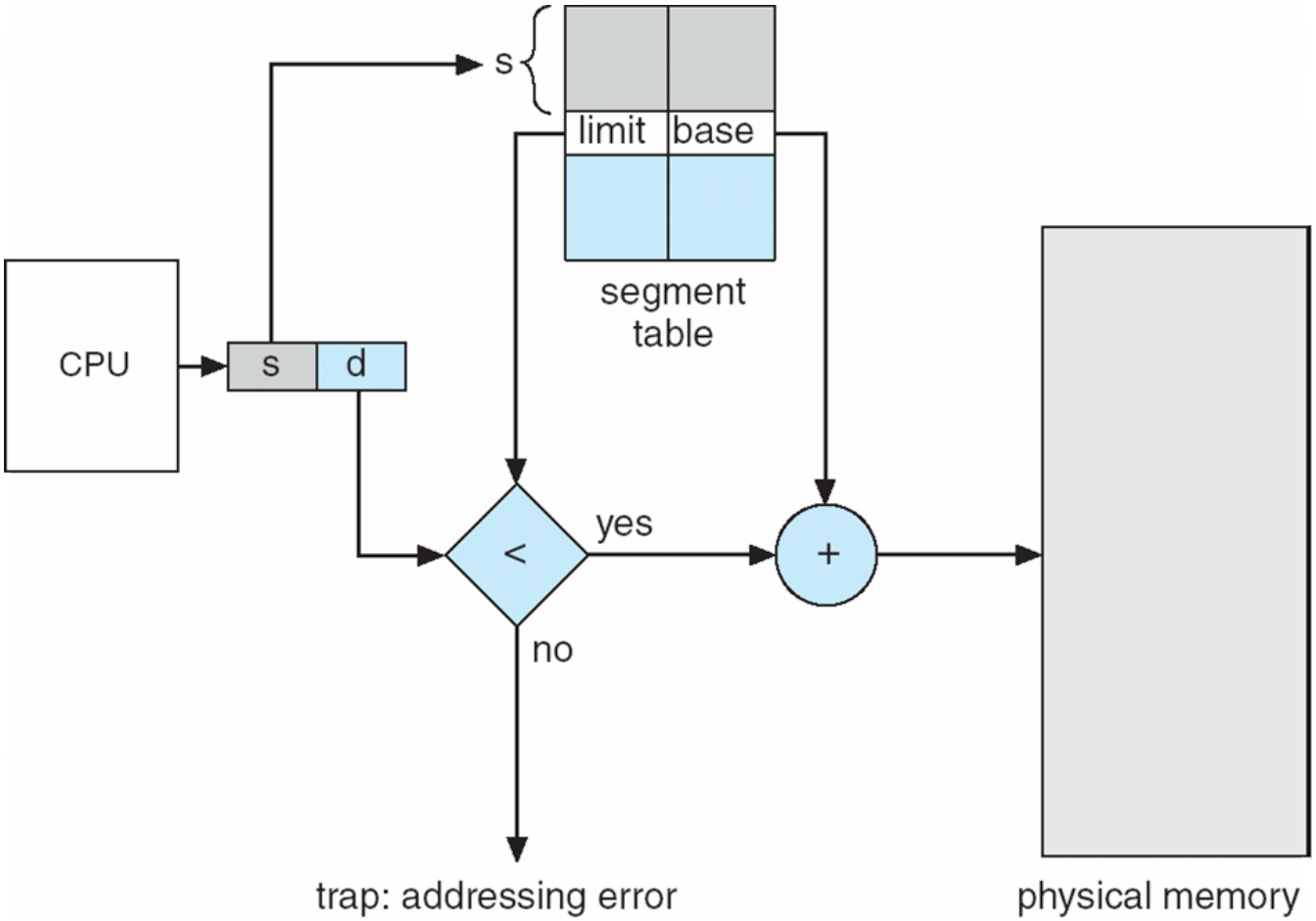
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

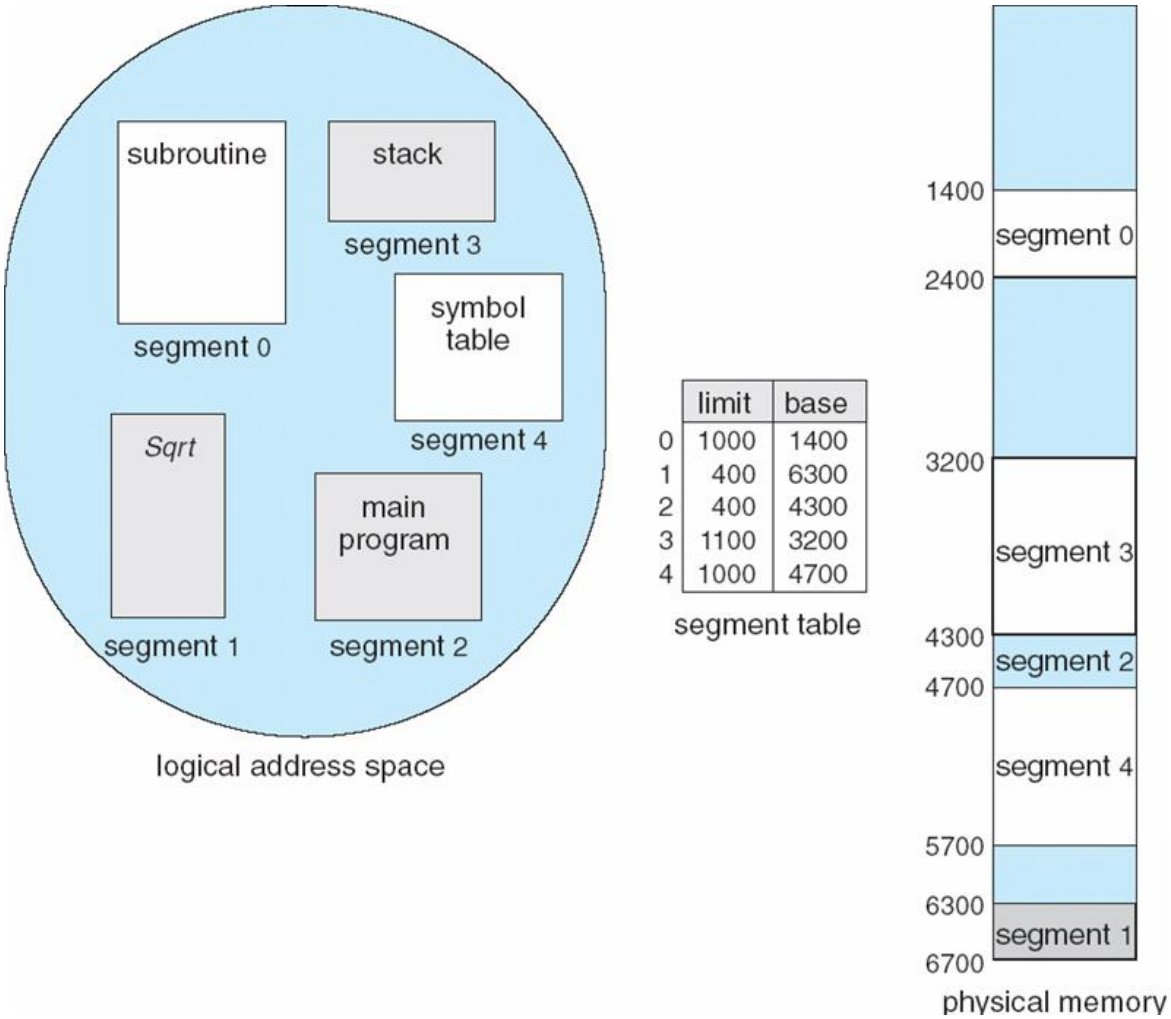
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ 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



Summary

- This section studied how page tables are implemented. It also introduced the concept of segmentation.