THE UNIVERSITY *of* EDINBURGH
informatics

Operating Systems (INFR10079)

2022/2023 Semester 2

Memory Management

(Paging)

abarbala@inf.ed.ac.uk

Chapter 9.3, 9.4, 9.6

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**)
  - Points to the page table, **part of CPU**
- **Page-table length register** (**PTLR**)
  - Indicates size of the page table, **part of CPU**

- Every data/code access requires **two memory accesses**
  - To *fetch the* ***page table entry*** (translation)
  - To *fetch the actual* ***memory content*** (data/code)

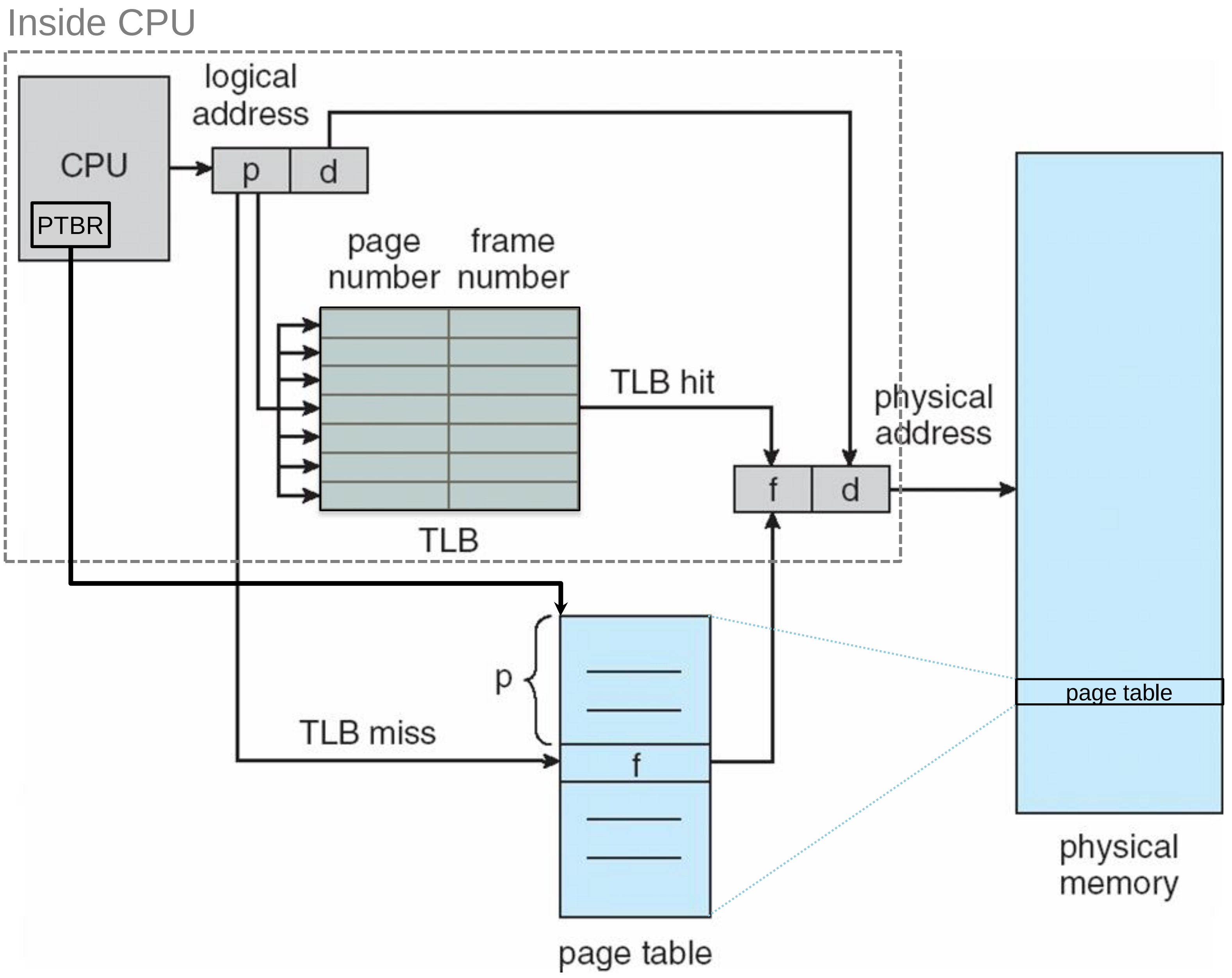
Implementation of Page Table: TLB

- How to avoid **two** memory accesses?
- **Translation look-aside buffers** (**TLBs**)
  - Fast-lookup hardware cache
    - Associative memory for parallel search
    - Entry of the form <page #, frame #>
  - Typically, small (e.g., 64 or 1,024 entries)

Page #	Frame #
0	14
2	18

- If translation
  - **Exists** in TLB (hit), use it (at no additional cost)
  - **Doesn't exist** in TLB (miss), fetch translation from memory
- Maintain translations for subsequent memory accesses
  - **Replacement policies** must be considered
  - Some entries can be **wired down** for permanent fast access

Paging Hardware With TLB



Effective Access Time (EAT) with TLB

- Memory access time
  - Time the CPU waits to access main memory directly
- Hit ratio = α
  - Percentage of times that a page number is found in TLB
- Miss ratio = $1 - \alpha$

- Consider α = 80%, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns

- Consider α = 99%, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

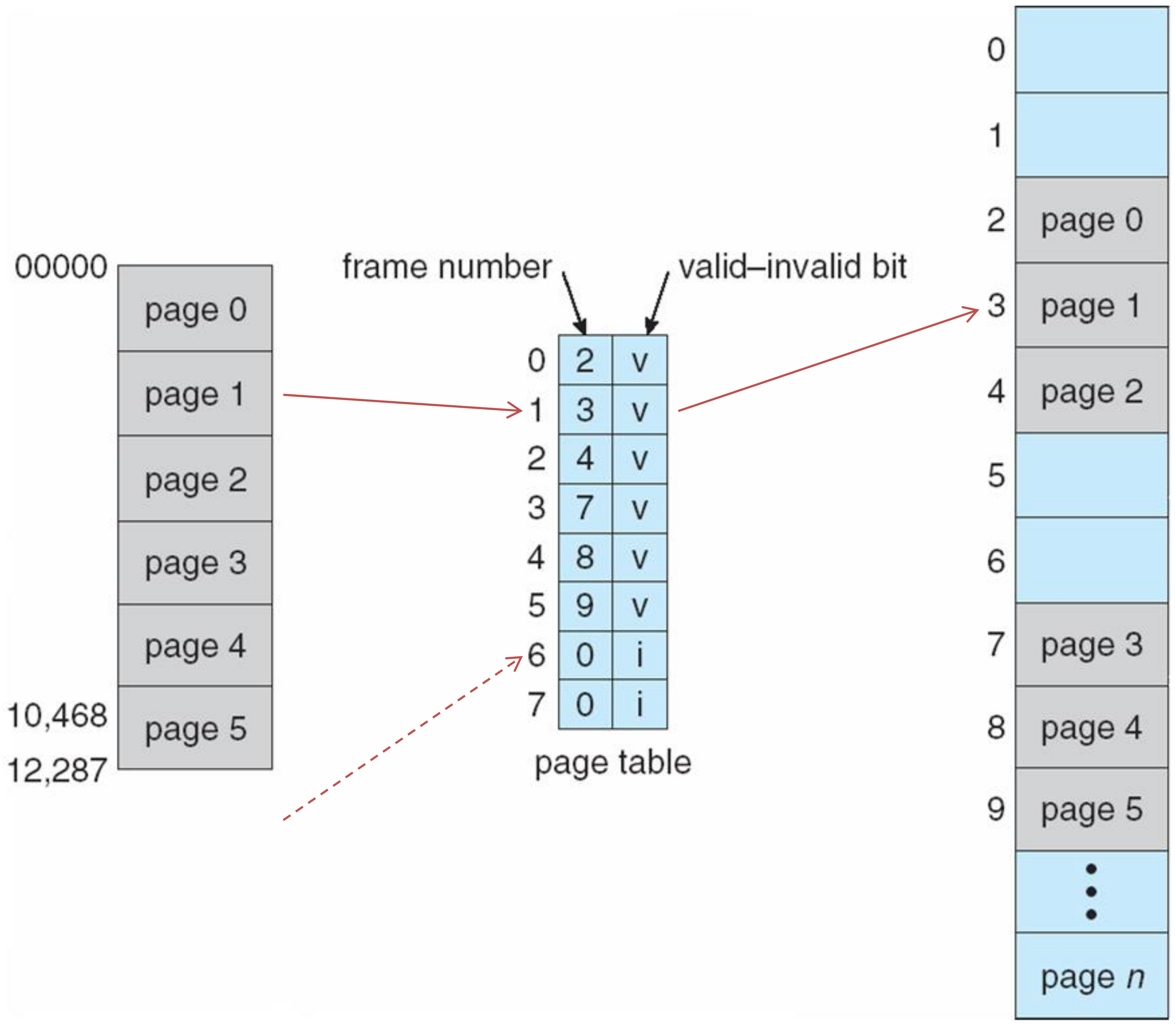
Memory Protection

- **Page Table Entries (PTEs)** contain more information

- Memory protection by **protection bits** in each PTE
  - What **type of access** is allowed?
    - Read-only, read-write, execute-only
  - Does a **translation exist**?
    - **Valid** indicates that the associated page
      - **is in the process' logical address space**, thus a legal page
    - **Invalid** indicates that the page
      - **is not in the process' logical address space**
  - …

- Any violations result in a trap to the OS kernel

Memory Protection: **Valid (v)** or **Invalid (i)** Bits



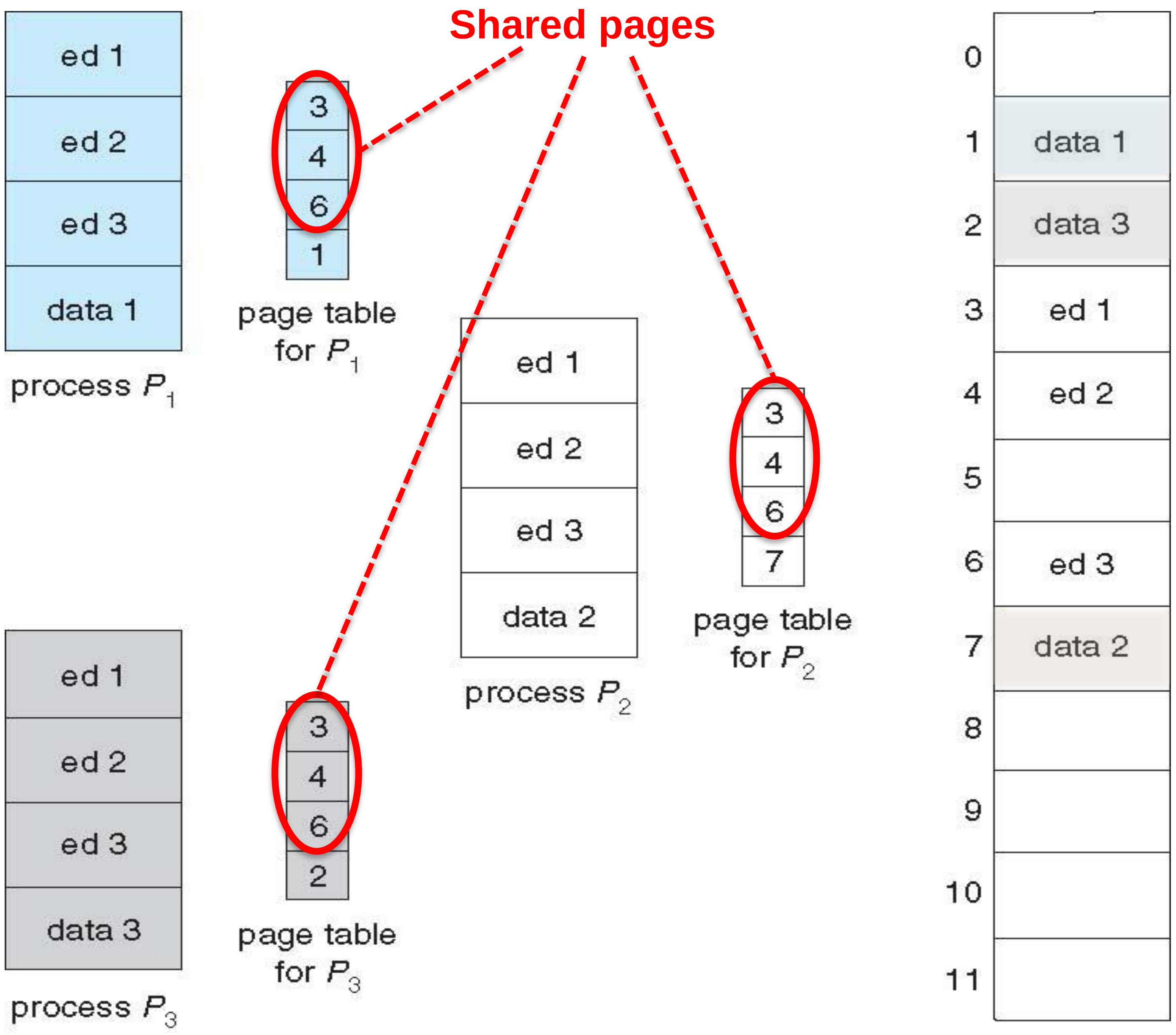
Shared Pages

- **Shared code or data**
  - **One copy** of read-only (reentrant) **code** shared among processes
    - Different processes may share the same library code
    - But they will have per-process data
  - Read-write **data pages** shared among processes for **communication**
    - Inter-process communication
  - **Somewhat different from multithreaded processes**
    - Different threads share the entire address space, and OS resources

- **Private code and data**
  - Each process keeps a separate copy of the code and/or data

- Pages for the private/shared code and data can appear **anywhere** in the logical address space

Shared Pages Example



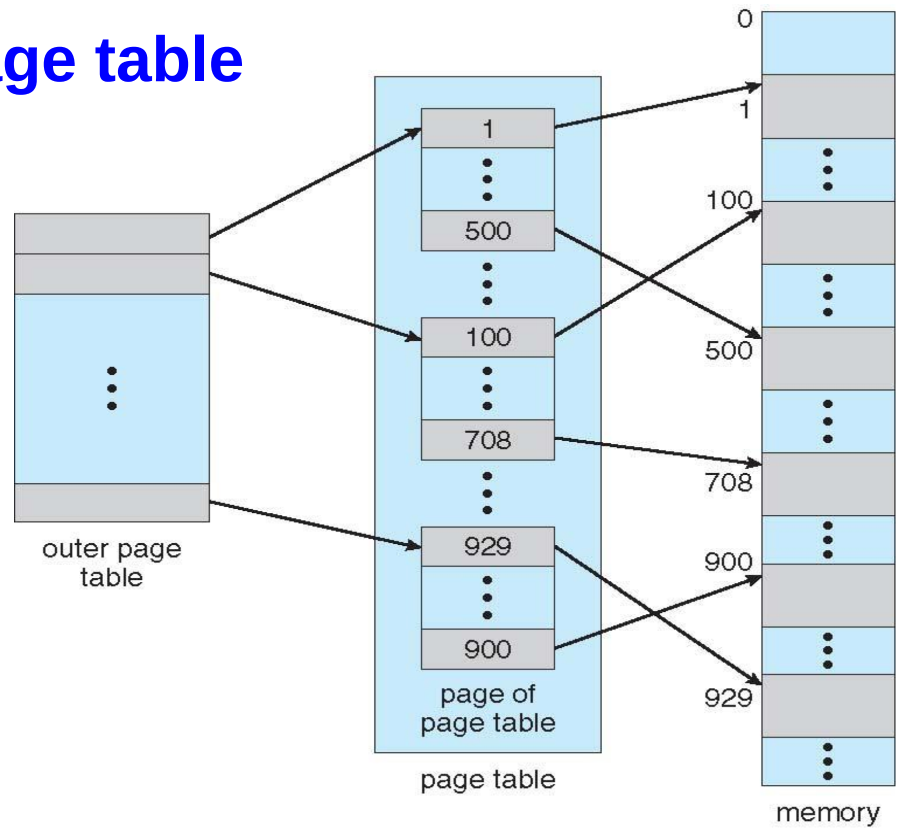
Structure of the Page Table

- Page table can **get huge**
  - Consider a 32-bit logical address space
  - Page size of 4 kB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - Each entry is 4 bytes, 4 MB of memory space required
    - Only for the page table, **independently of translations**
    - Need to allocate **contiguously** in physical memory
    - Costs a lot for small memories
      - 0.1% of 4GB per process

- Alternative **constructions**
  - **Hierarchical** Page Tables
  - **Hashed** Page Tables
  - **Inverted** Page Tables

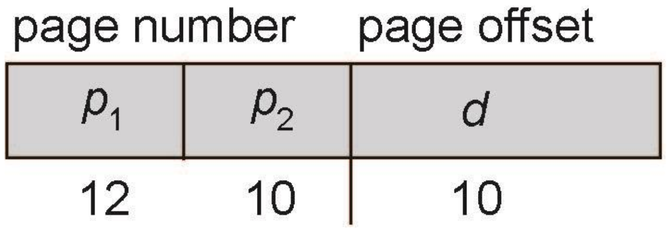
Hierarchical Page Tables

- Break up the entire logical address space into **multiple** page tables
- Then construct **another table** that refers to each of such page tables
- **Two-level page table**



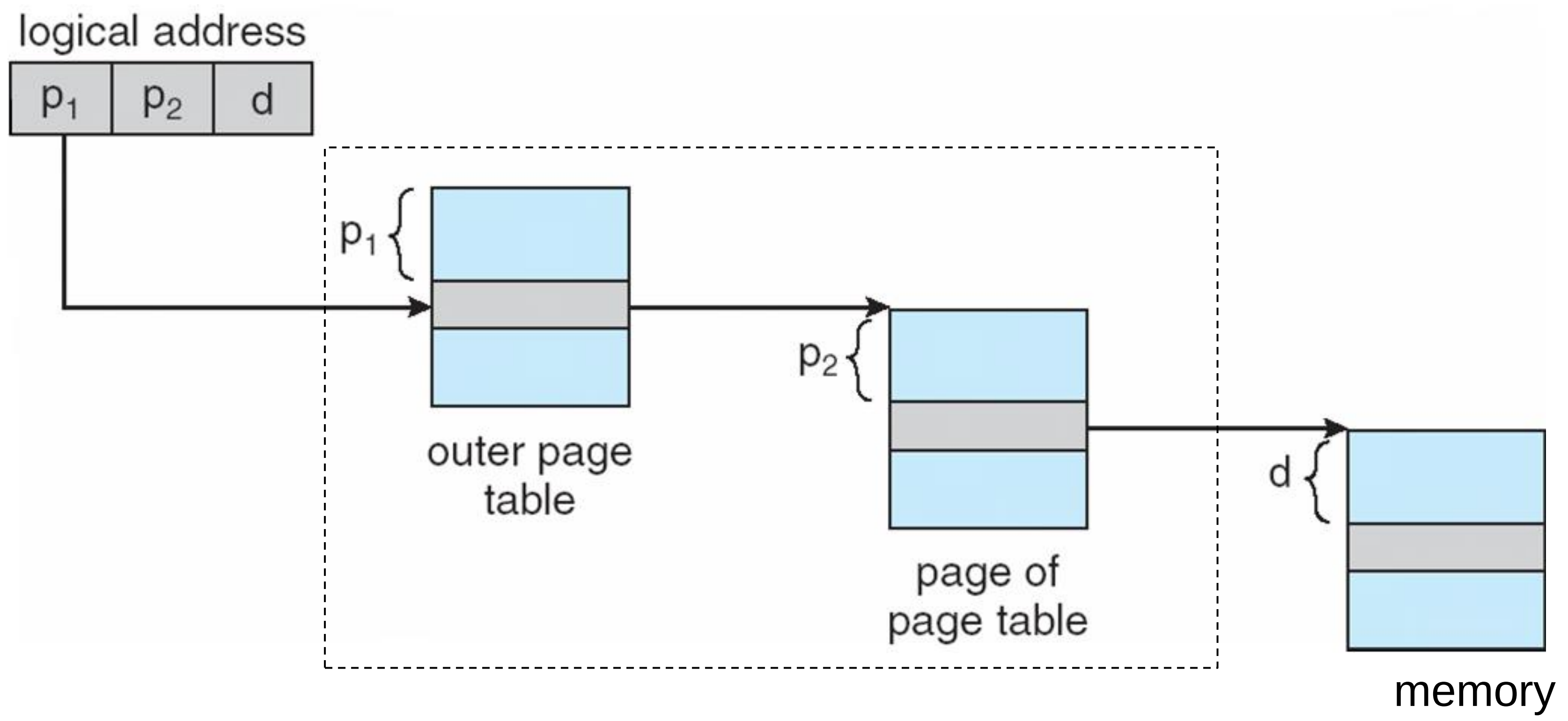
Two-Level Paging Example

- Logical address (32-bit machine with 1K page size) divided into
  - a **page number** consisting of 22 bits
  - a **page offset** consisting of 10 bits
- Page table is paged, the page number is further divided into
  - 12-bit **outer** page number
  - 10-bit **inner** page number
- A logical address looks like



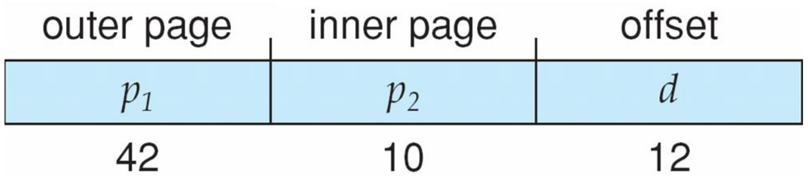
- p_1 is an index into the outer page table
- p_2 is an index within the page of the inner page table

Two-Level Paging Address-Translation Scheme

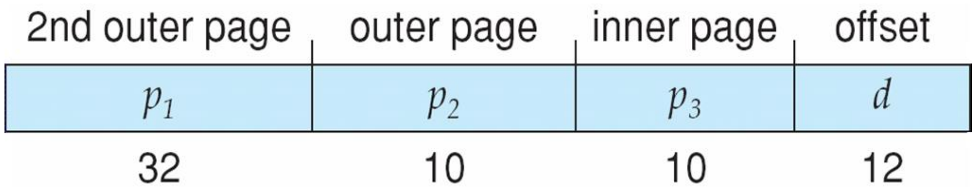


64-bit Logical Address Space

- How many levels?
- If page size is 4 kB (2^{12}) and **single-level** page table
  - Page table has 2^{52} entries
- If **two-level** scheme
  - Inner page tables could be 2^{10} entries
  - Outer page table has 2^{42} entries (4,398,046,511,104)



- If **three-level** scheme – add a 2nd outer page table
  - The 2nd outer page table is still 2^{32} entries (4,294,967,296)



Multi-level Paging Scheme

- On 64bit machines logical address space < 2^{64}
  - For practical reasons (no machine with 18 zetta bytes)
- But 4-level and 5-level page tables exist

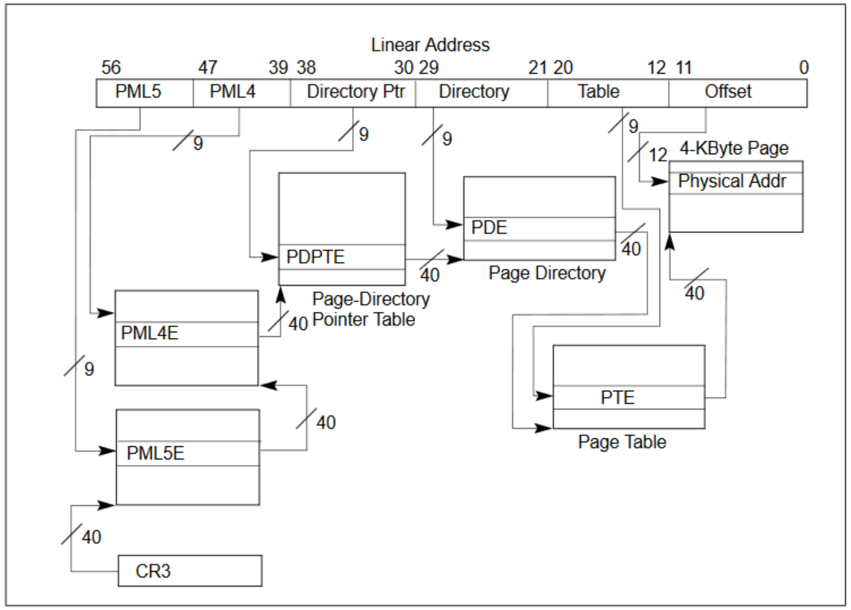


Figure 2-1. Linear-Address Translation Using 5-Level Paging

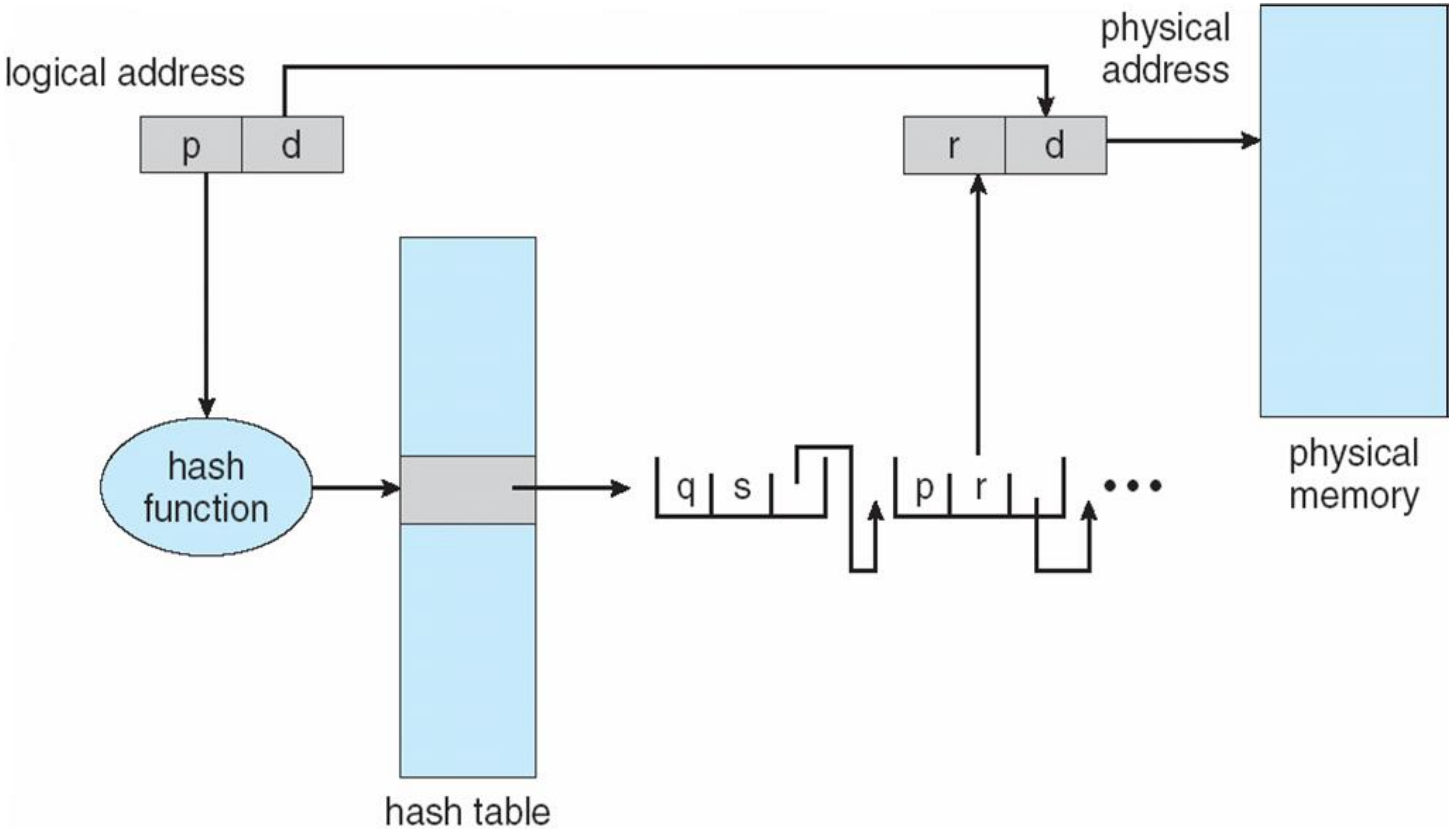
https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf

Hashed Page Tables

- The **logical page number** is hashed into a page table index
- Each entry **chains** elements hashing to the same location
- Each element **contains**
  - Logical page number
  - Value of the mapped page frame
  - Pointer to the next element
- Logical page numbers are compared searching for a **match**
  - If a match is found, the corresponding physical frame is extracted

- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages
    - 16 rather than 1

Hashed Page Table Example



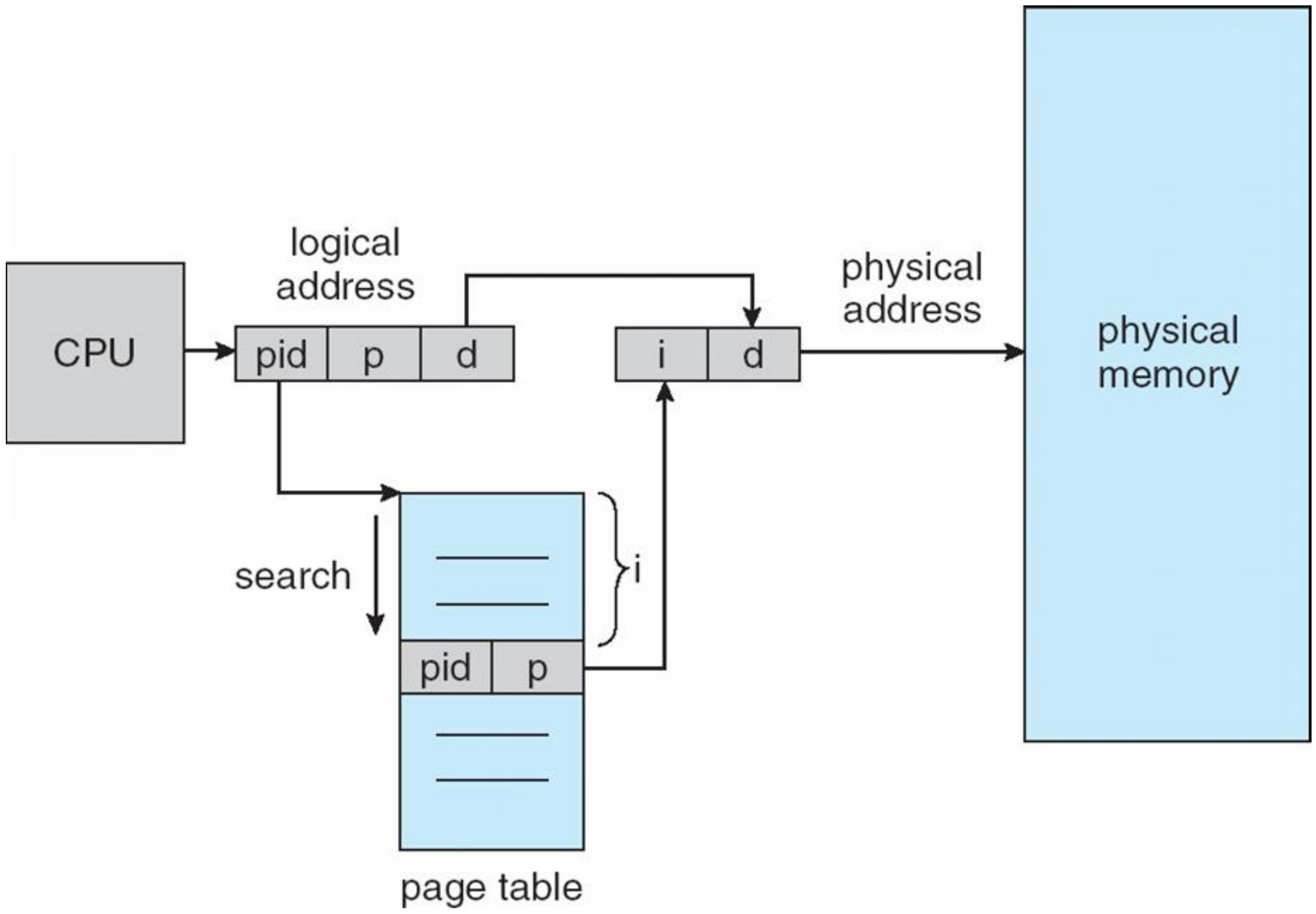
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages
  - **track all physical pages**

- One entry for each physical page of memory
- Entry consists of
  - **Virtual address** of the page stored in physical memory location
  - Information about the **process that owns** the page (protection)
- **Decreases memory** needed to store translations
  - Increases time needed to search the table
  - Use hash table to limit the search to one/few page-table entries

- How to implement shared memory?
  - Need OS intervention

Inverted Page Table Architecture



Why Paging? #1

- Inter-process memory protection
  - Process A address XYZ is different from Process B XYZ
- Protect code from being rewritten
  - Code pages read-only
  - A bad pointer can't change the program code
- Detect null pointer dereferencing at runtime
  - First page of the logical address space invalid
  - References to address 0 cause trap to OS
- Reduce memory usage (shared libraries)
  - Have one copy of a library in physical memory, not one per process
  - All page table entries to library point to the same set of physical frames
- Generalize use of "shared memory"
  - Regions of two processes' address spaces map to the same frames

Why Paging? #2

- **Copy-on-write (CoW)**, e.g., on fork()
  - Instead of copying all pages, create shared mappings of parent pages in child address space
    - Make shared mappings read-only for both processes
    - When either process writes, fault occurs, OS “splits” the page
- **Memory-mapped files**
  - Instead of using open, read, write, close
    - “map” a file into a region of the logical address space
      - e.g., into region with base ‘X’
    - Accessing logical address ‘X+N’ refers to offset ‘N’ in file
    - Initially, all pages in mapped region marked as invalid
  - OS reads a page from file whenever invalid page accessed
  - OS writes a page to file when evicted from physical memory
    - Only necessary if page is dirty

Summary

- Non-contiguous (physical) memory allocation
  - Reduced or no fragmentation
  - Enable sharing
  - Require hardware support

- Segmentation
  - Variable size chunks
- Paging
  - Fixed size chunks
  - Multiple optimizations