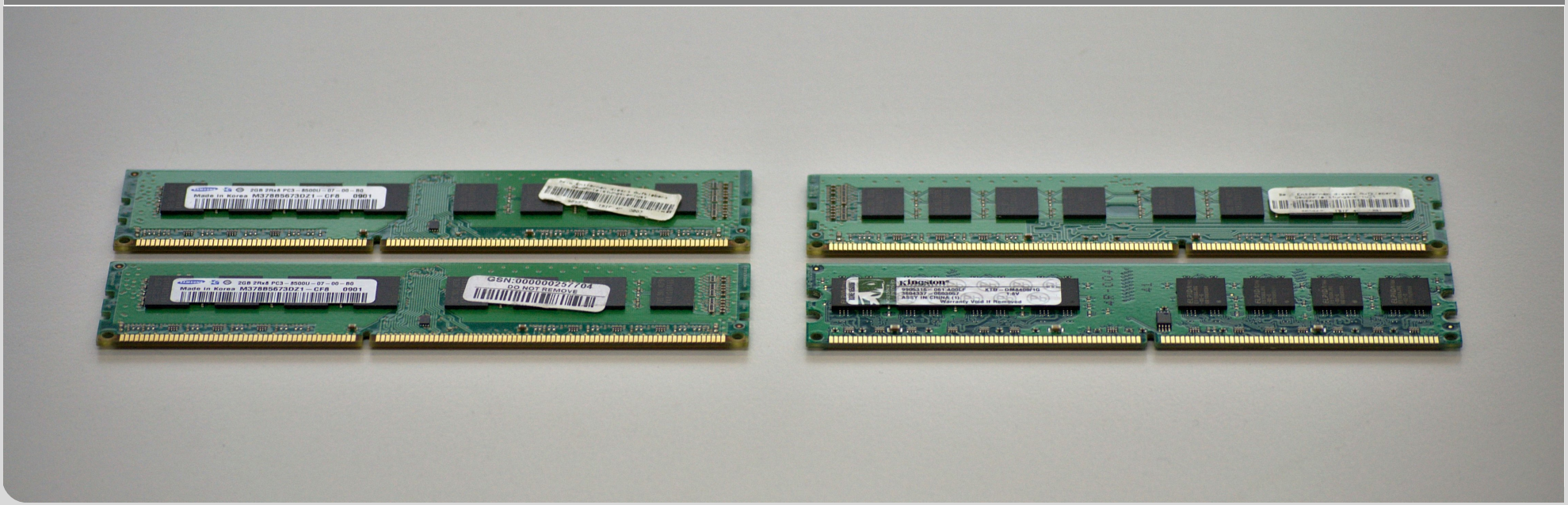
Karlsruhe Institute of Technology

Betriebssysteme

10. Paging

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP



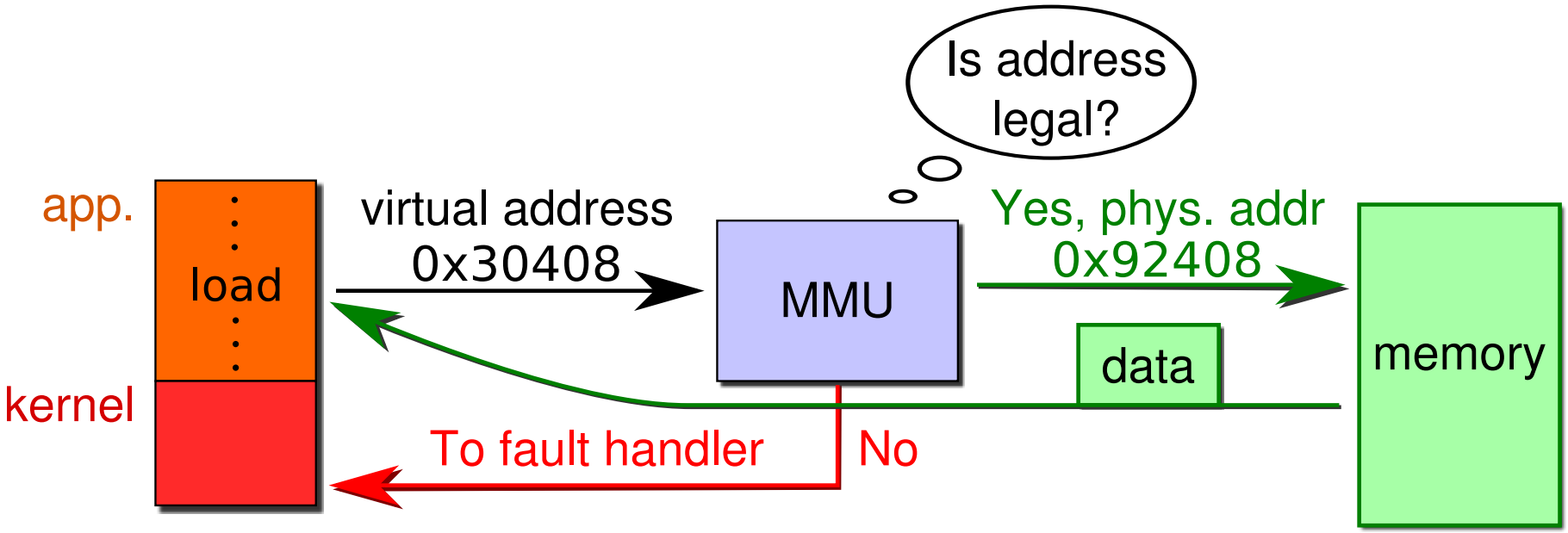
Desired properties when sharing physical memory

- Protection
  - A bug in one process must not corrupt memory in another
  - Don't allow processes to observe other processes' memory (pgp/ssh-agent)
- Transparency
  - A process shouldn't require particular physical memory addresses
  - Processes should be able to use large amounts of contiguous memory
- Resource exhaustion
  - Allow that the sum of sizes of all processes is greater than physical memory

Need protection and (dynamic) relocation

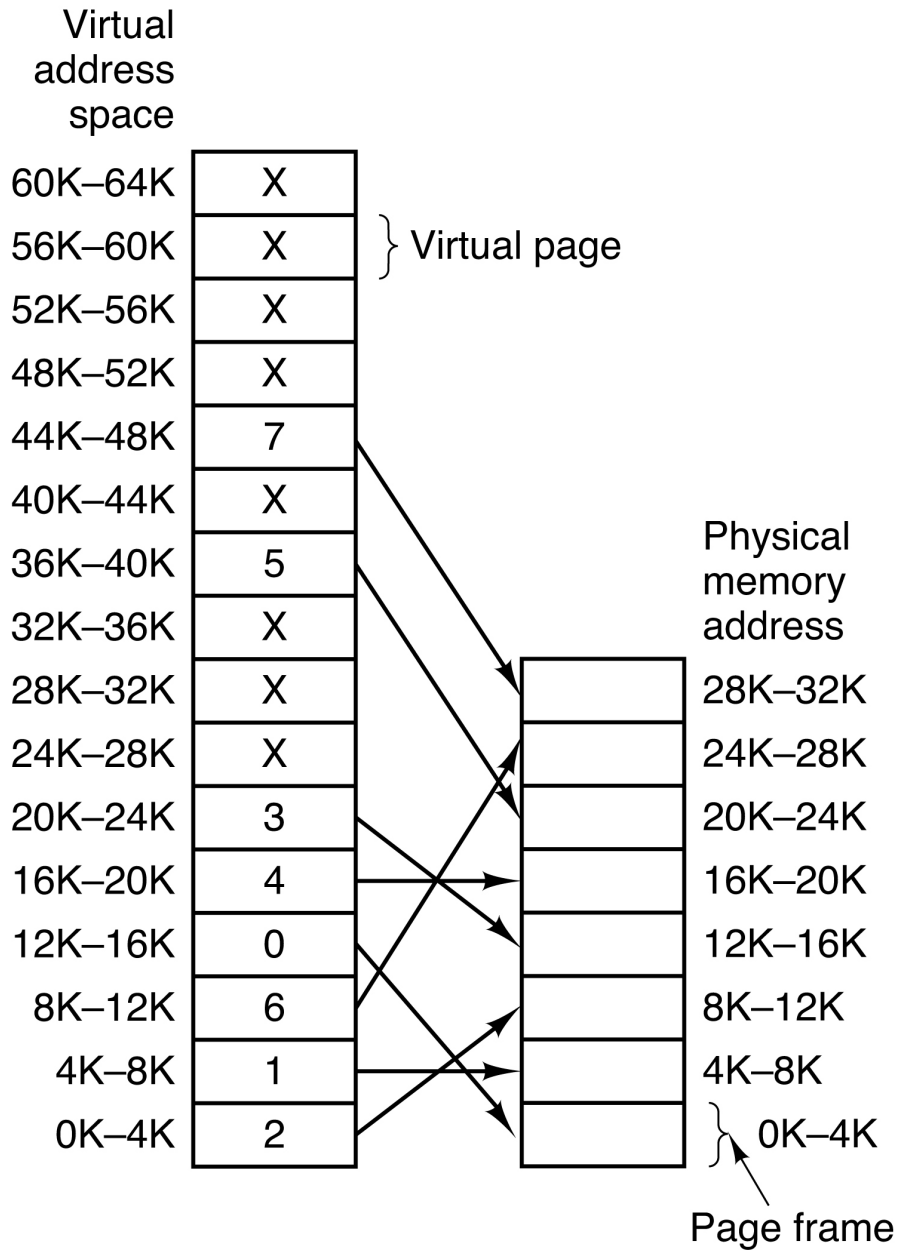
Memory-Management Unit (MMU)

- Need hardware support to achieve safe and secure protection
- Hardware device maps virtual to physical address
- The user program deals with virtual addresses
  - It never sees the real physical addresses



Paging

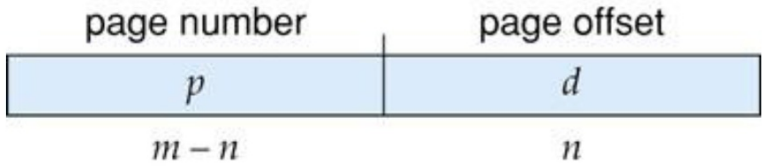
- A Present Bit in the page table indicates if a virtual page is currently mapped to physical memory
- MMU reads the page table and autonomously translates valid mappings
- If a process issues an instruction to access a virtual address that is currently not mapped, the MMU calls the OS to bring in the data (page fault)

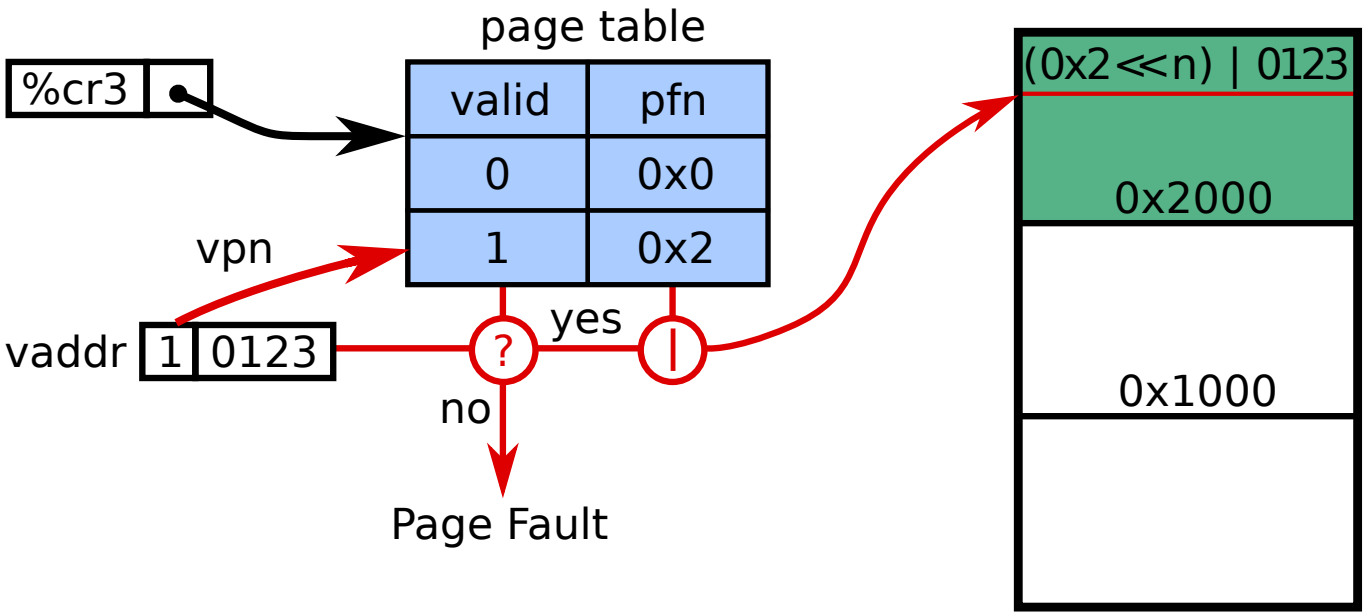


Page Table Layouts

Linear Page Table

- Virtual address is divided into:
  - Virtual page number: Index into the page table which contains base address of each page in physical memory
  - Page offset: Concatenated with base address results in physical address



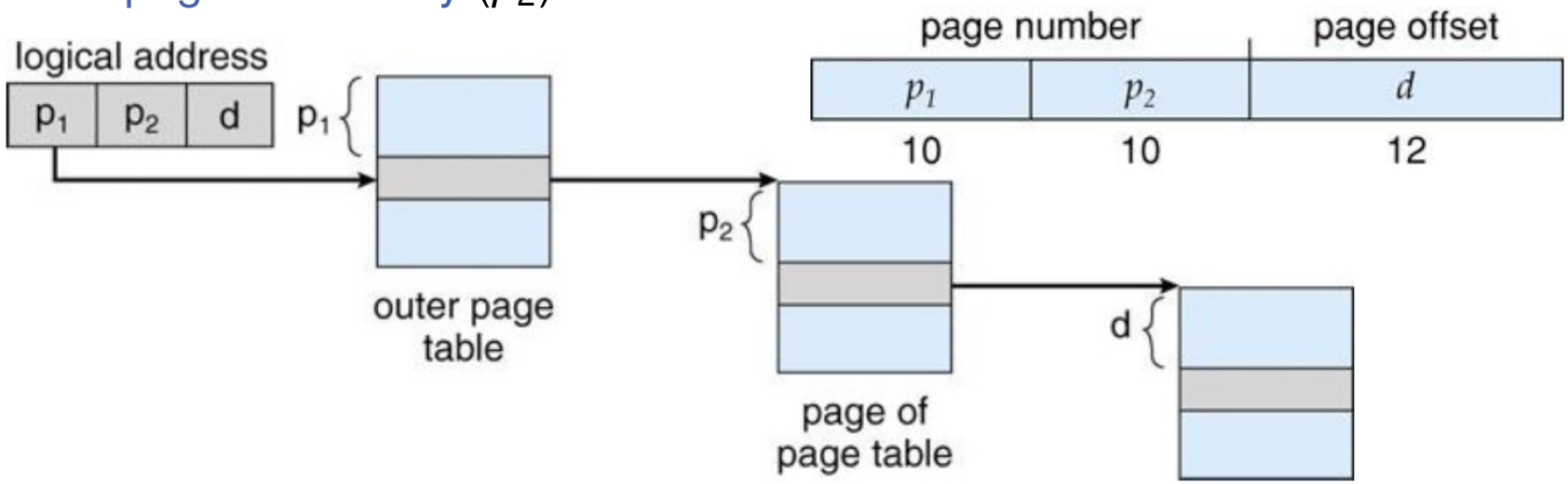


Hierarchical Page Table

- Problem: For every address space, need to keep complete page table in memory that can map all **virtual** page numbers (vpn)
- Idea: Don't need complete table
  - Most virtual addresses are not used by process
  - Unused vpns do not have a valid mapping in the page table
    → No need to store invalid vpns
- Another level of indirection saves the day:
  - Subdivide the virtual address into multiple page table indices forming a hierarchical page table

Example: Two-Level Page Table

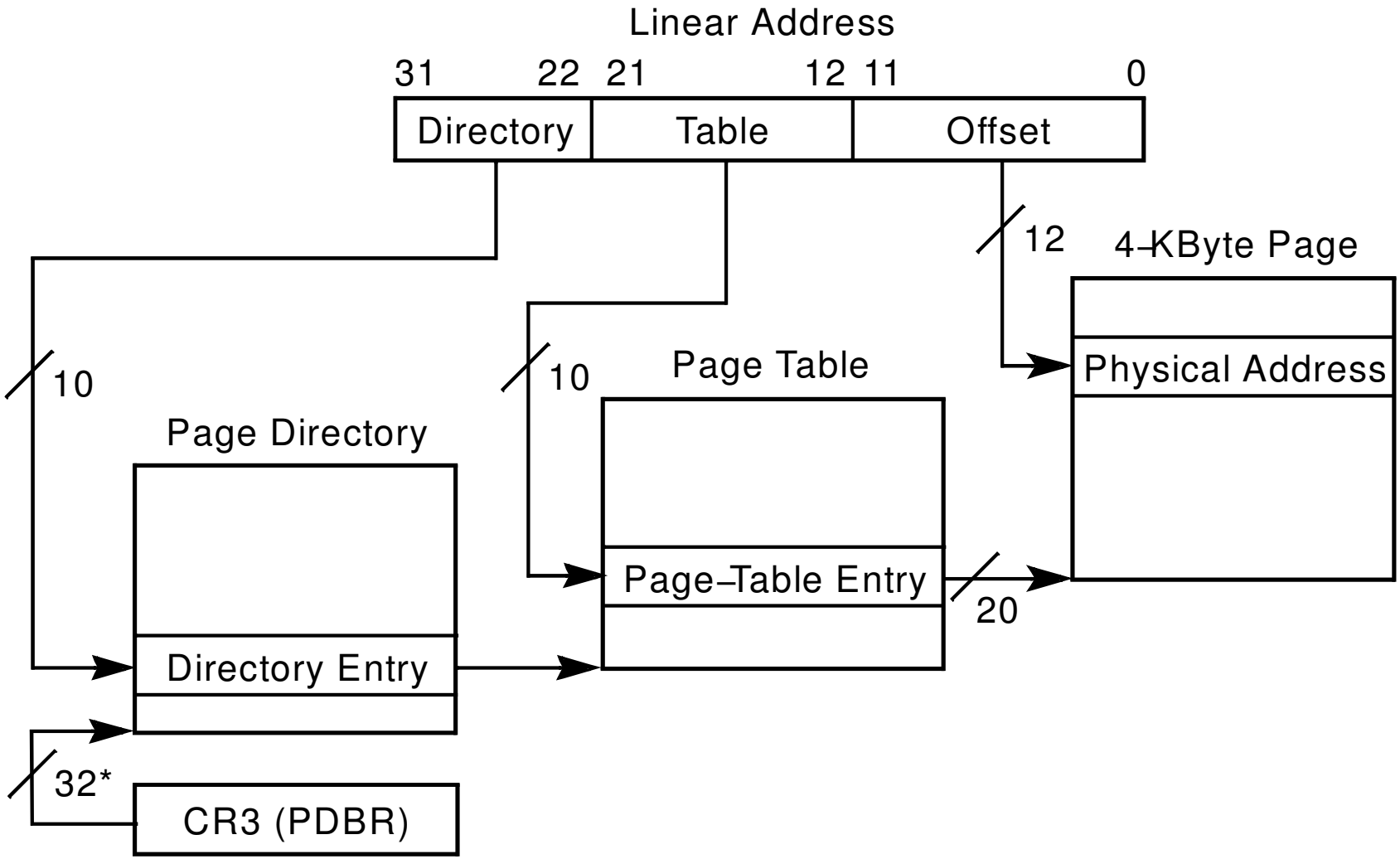
- On a 32-bit machine with 4-KiB pages, divide the virtual address into:
  - Page number (p): 20 bits
  - Page offset (d): 12 bits
- The page table itself can be paged, to save memory, subdivide the vpn:
  - Index in page directory (p_1): 10 bits
  - Index in page table entry (p_2): 10 bits



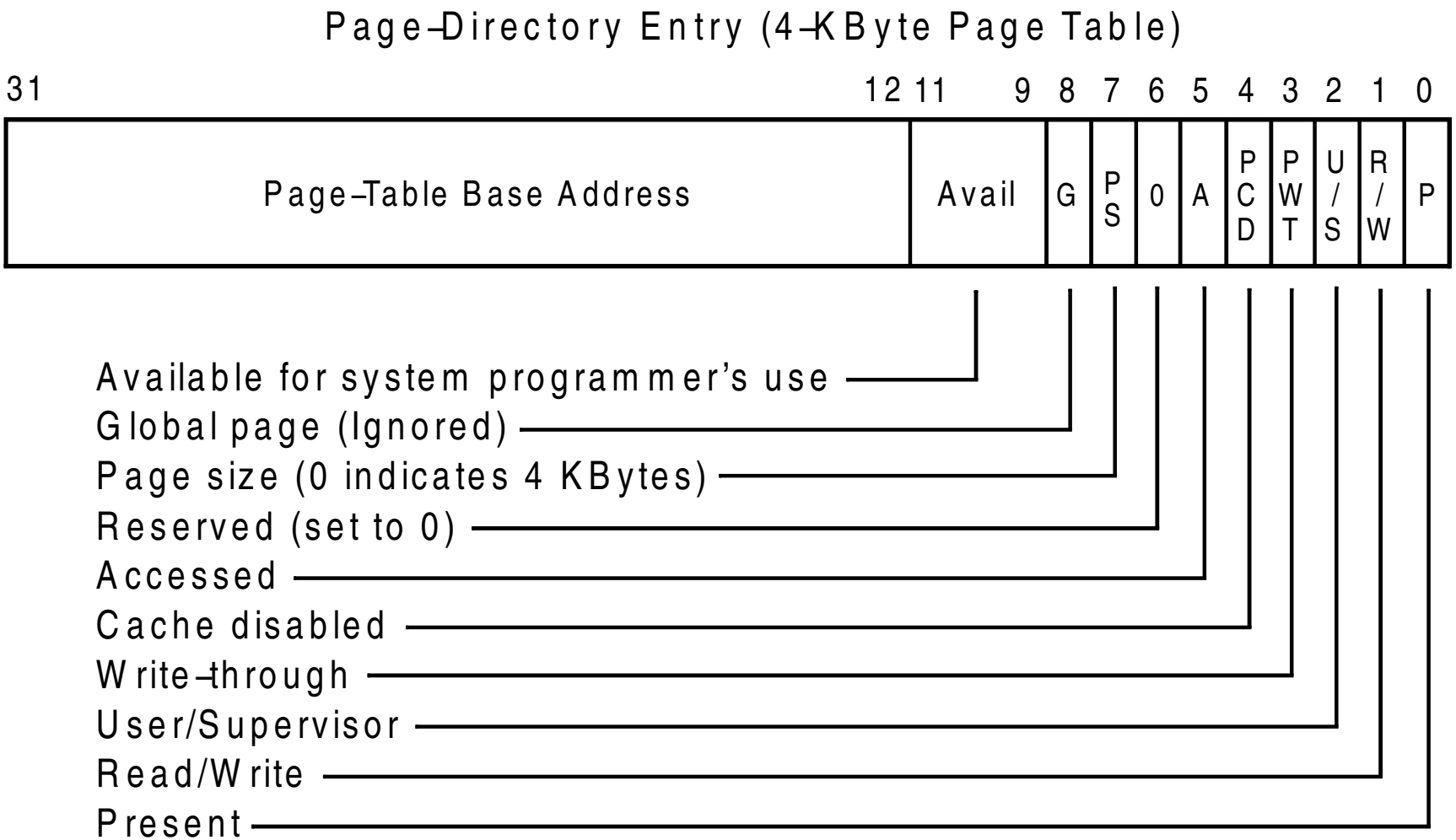
- For ranges of 1024 invalid pages, reset present bit in page directory
  - Save space of second-level page table

Example: 32 Bit Intel Architecture (IA-32) – Page Table

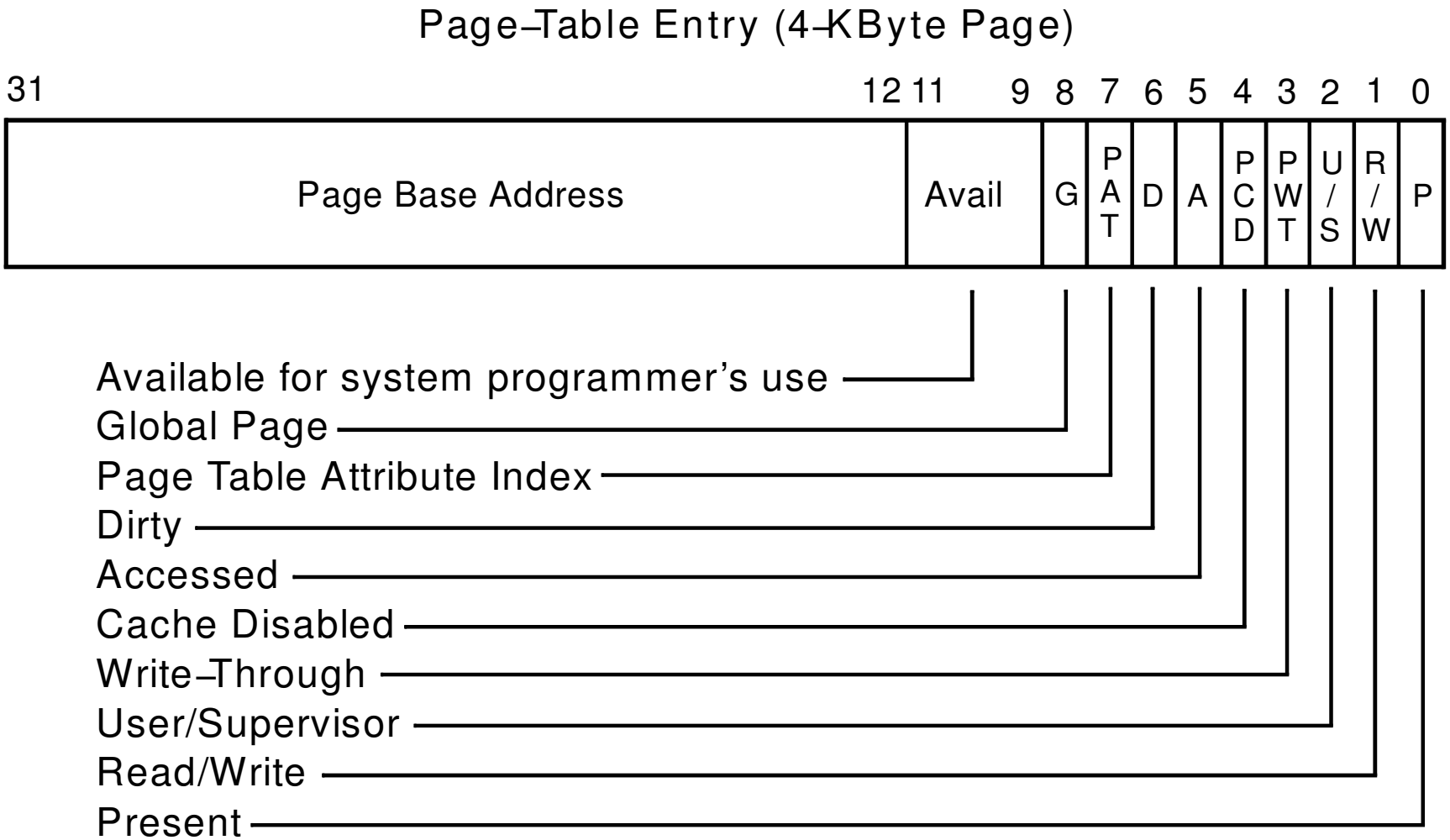
- Typical page table illustration from the Intel Architecture Manual



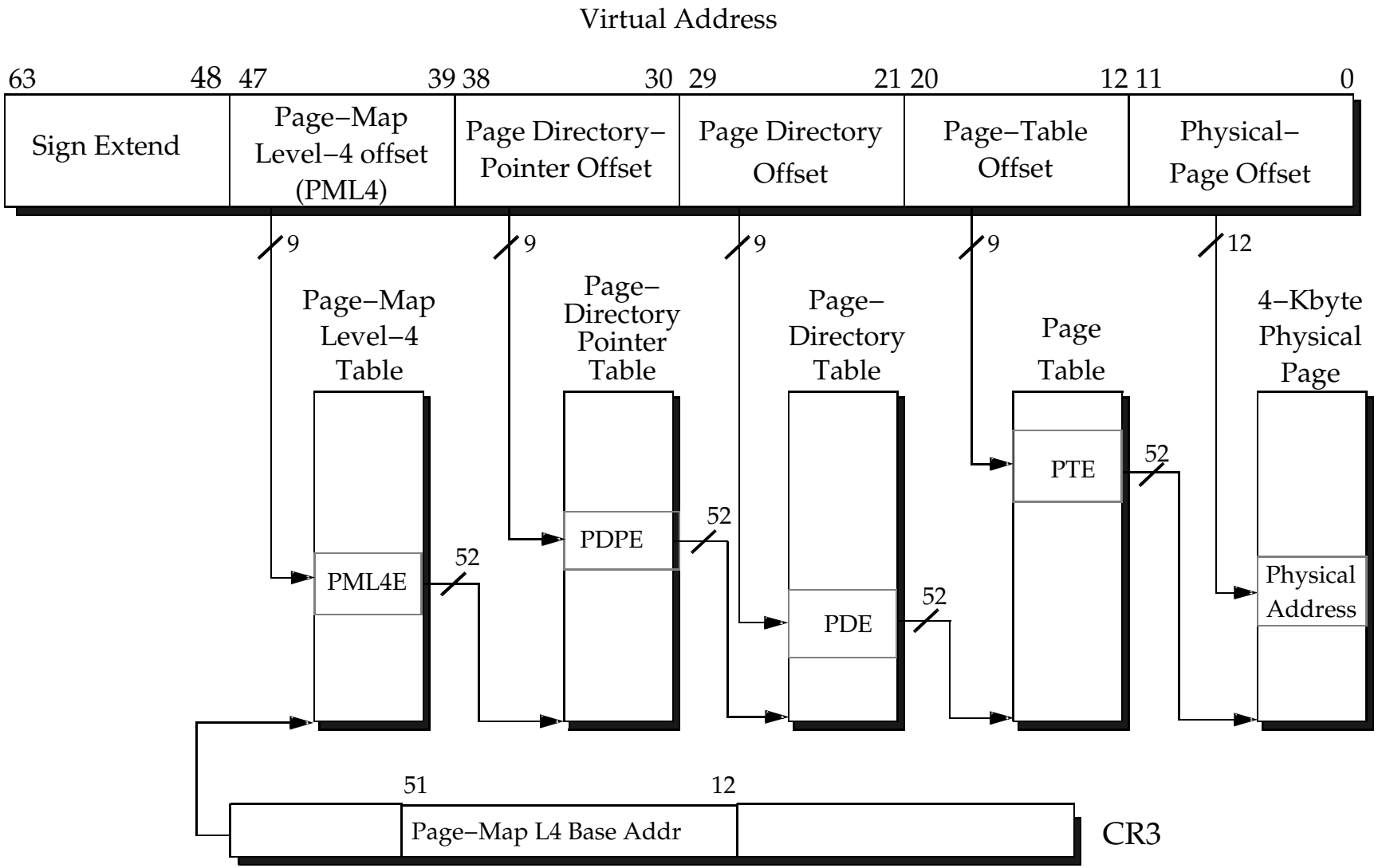
Example: IA-32 – Page Directory



Example: IA-32 – Page Table Entry



Example: Four-Level Page Table (x86-64)

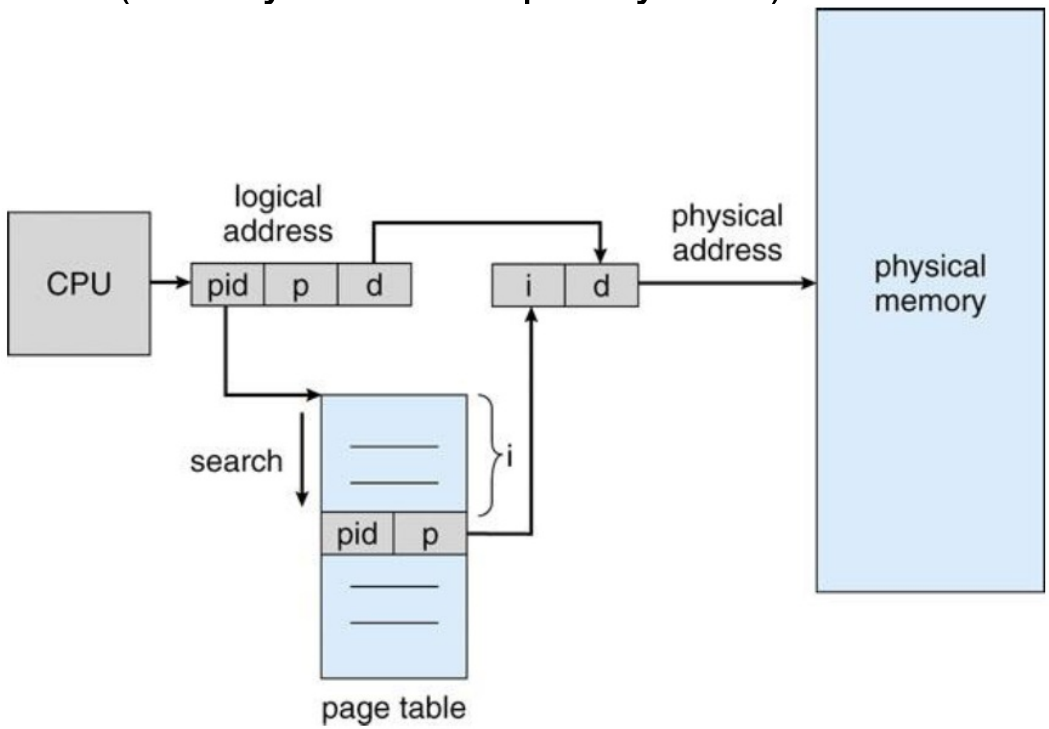


Linear Inverted Page Table

- Problem: AS large (64-bit) but only few virtual addresses are mapped
  → Much memory wasted on page tables in the system
  → Lookup slow due to many levels of hierarchy
- Possible Solution: Invert page table mapping
  - Map physical frame to virtual page instead of the other way around
  - Single page table for **all processes** (exactly one table per system)
  - One page table entry for each physical page frame

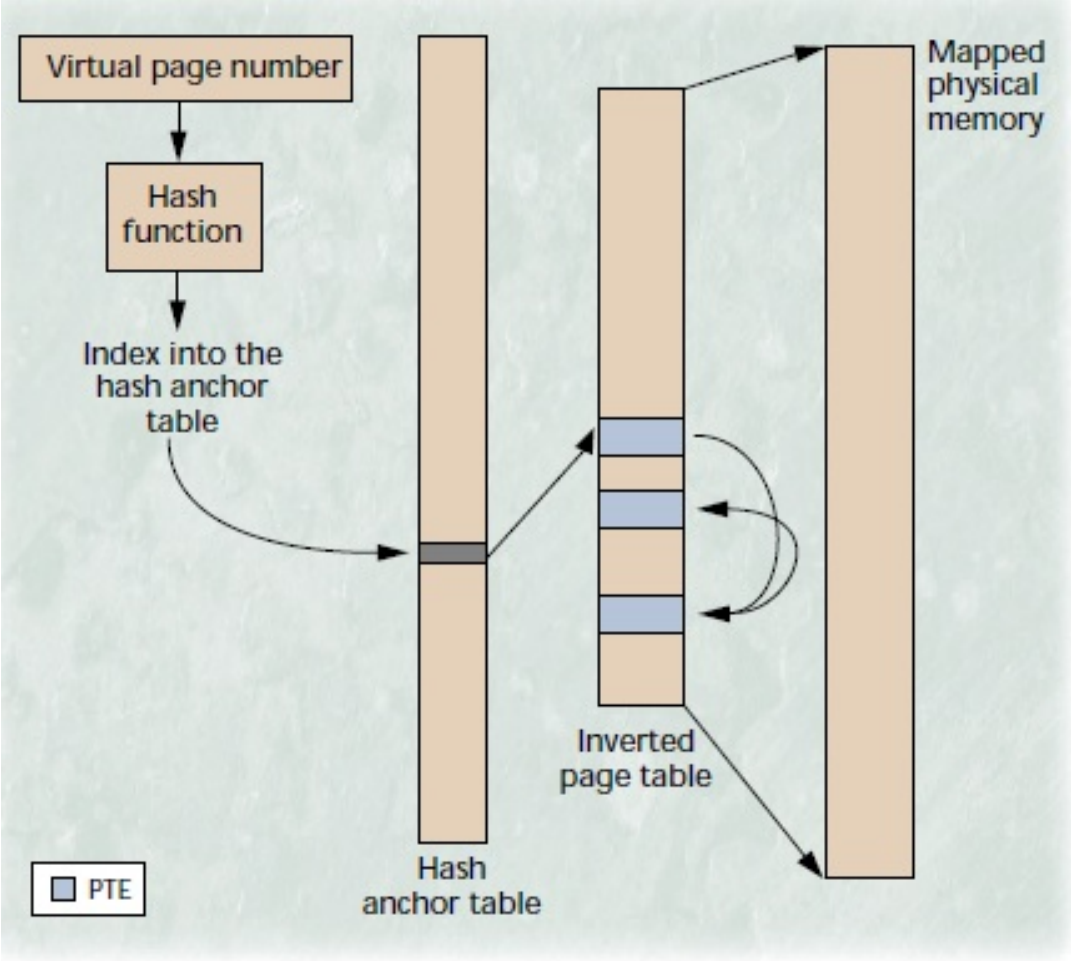
\+ Less overhead for page table meta data

– Increases time needed to search the table when a page reference occurs

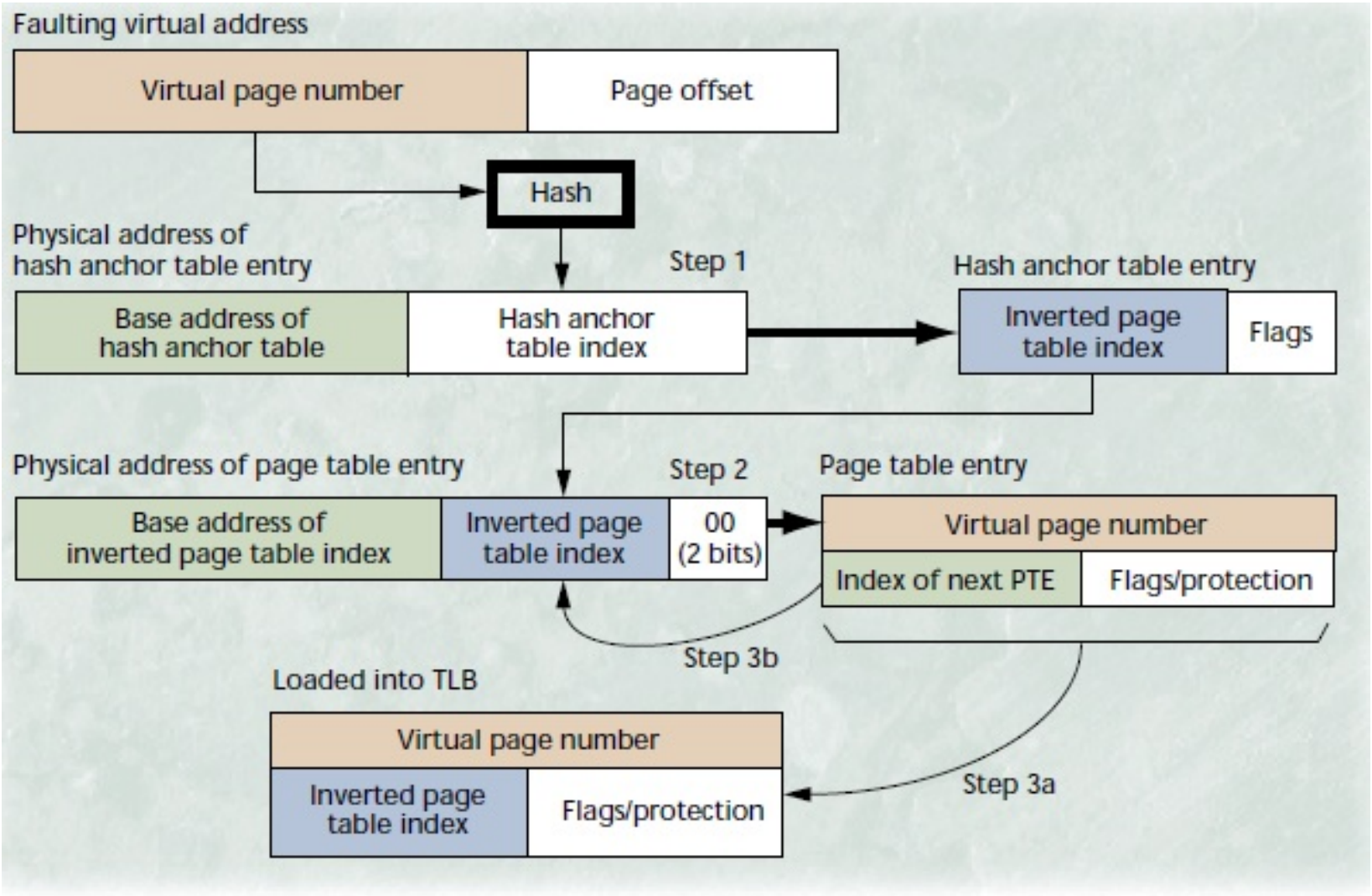


Hashed Inverted Page Table

- Hash anchor table limits the search to at most a few page-table entries (e.g., PPC, PA-RISC)



Hashed Inverted Page Table Lookup



Translation Lookaside Buffer

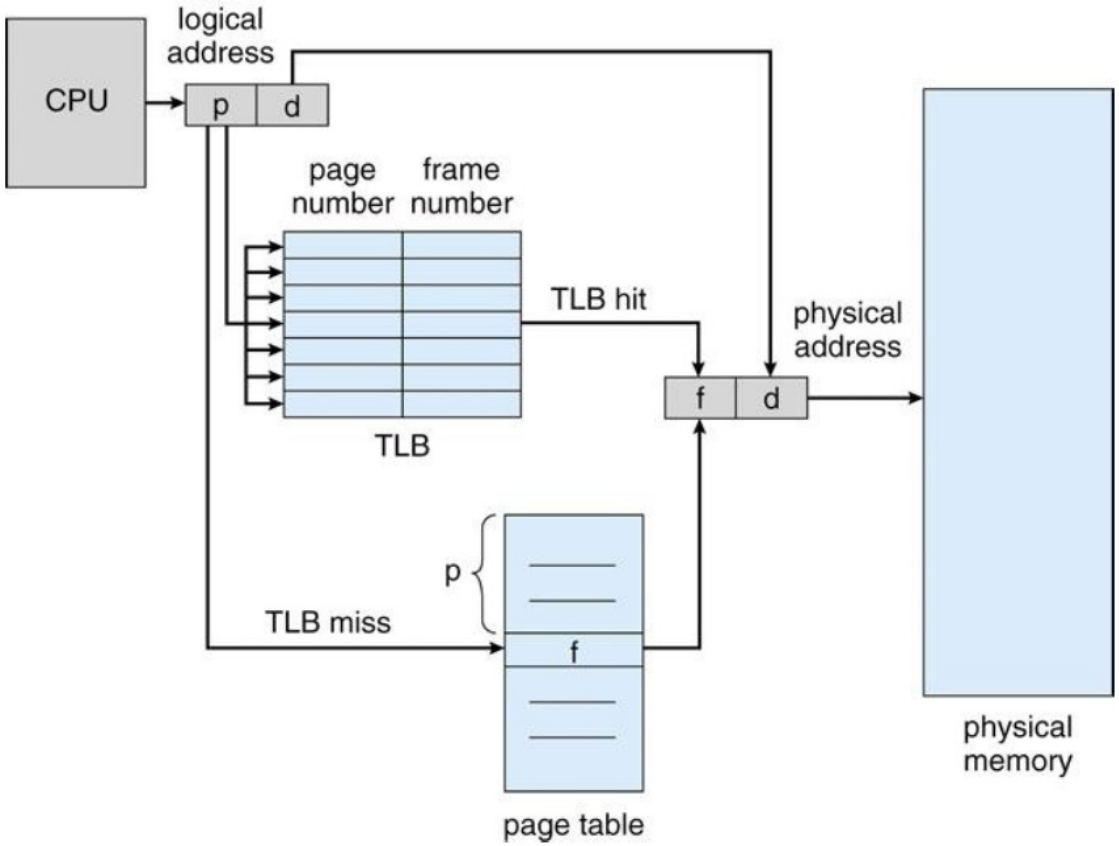
Making Paging Fast

Naïve Paging is Slow

- Every load/store requires multiple memory references
- 4-Level Hierarchy: 5 memory references for every load/store
  - 4 references to page directories and tables
  - 1 access to data
- Idea: Add a cache that stores recent memory translations
  - Translation Lookaside Buffer (TLB) maps <vpn> to <pfn, protection>
  - Typically: 4-way to fully associative hardware cache in MMU
  - Typically: 64 - 2K entries
  - Typically: $\sim$ 95%–99% hit rate

TLB Operation

- On every load/store
  - Check if translation result is already cached in TLB (TLB hit if available)
  - Otherwise walk page tables and insert result into TLB (TLB miss)
- Quick: Can compare many TLB entries in parallel in hardware



TLB Miss

- Need to evict an entry from TLB on TLB miss
- Need to load an entry for the missing virtual address into the TLB
- TLBs can be software-managed or hardware-managed
  - Depends on the architecture
- Software-managed TLBs
  - OS receives TLB miss exception
  - OS decides which entry to evict (drop) from TLB
  - Generally, OS walks page tables in software to fill new TLB entry
  - TLB entry format specified in instruction set architecture (ISA)
  - e.g., MIPS uses software-managed TLB
- Hardware-managed TLBs
  - Evict a TLB entry based on a policy encoded in hardware without involving the OS
  - Walk page table in hardware to resolve address mapping
  - e.g, x86-64 and ARM use hardware-managed TLB

Address Space Identifiers

- Problem: vpn is dependent on AS
  - vpns in different AS can map to different pfns
  - Need to clear TLB on AS switch
- Idea: Solve vpn ambiguity with additional identifiers in the TLB
- TLB with Address Space Identifier (ASID) in every entry
  - Map <vpn, ASID> to <pfn, protection>
  - Avoids TLB flush at every address-space switch
- Results in less TLB misses
  - Some TLB entries are still present from the last time the process ran

TLB Reach

- TLB Reach (a.k.a. TLB Coverage): The amount of memory accessible with TLB hits
  - TLB Reach = (TLB Size) $\times$ (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of TLB misses
- Increase the Page Size
  - \+ Need fewer TLB entries per memory
  - – Increases internal fragmentation
- Provide Multiple Page Sizes (e.g., use normal pages and hugepages)
  - \+ Allows applications that map larger memory areas to increase TLB coverage with minimal increase in fragmentation
- Increase TLB size
  - – Expensive

Effective Access Time

- Associative lookup takes τ time units
  - e.g., τ = 1 ns
- A memory cycle takes μ time units
  - e.g., μ = 100 ns
- TLB hit ratio α
  - Percentage of all memory accesses whose translation is already cached in the TLB
  - e.g., α = 99%
- Effective Access Time (EAT) for linear page table without cache

$$EAT = (\tau + \mu) \cdot \alpha + (\tau + 2 \cdot \mu) \cdot (1 - \alpha) = \tau + 2 \cdot \mu - \mu \cdot \alpha$$

Impact of Program Structure on TLB-Miss Overhead

- `uint32_t data[128][128];`
  - Each row is stored in one page (e.g., 512 bytes page size)

Program 1

```
for( j = 0; j < 128; j++ )
   for( i = 0; i < 128; i++ )
      data[i][j] = 0;
```

$128 \times 128 = 16,384$ TLB misses

- Program 1 iterates
  - first byte on each page
  - second byte on each page
  - ...

Program 2

```
for( j = 0; j < 128; j++ )
   for( i = 0; i < 128; i++ )
      data[j][i] = 0; // swapped
```

128 TLB misses

- Program 2 iterates
  - each byte of first page
  - each byte of second page
  - ...

Summary

- Page tables communicate between OS and MMU hardware
  - How virtual addresses in each address space translate to physical addresses
  - Which kind of accesses the MMU should allow or signal to the OS (e.g., write to read-only pages)
- Different page table layouts have been developed in the past
  - Linear page table
  - Hierarchical page tables
  - Inverted page tables
  - Hashed page tables
- Performing page table lookups for every memory access significantly slows down the execution of programs
  - The translation lookaside buffer (TLB) caches page table lookups that have previously been performed to speed up the memory translation process
  - Typical TLBs cover 95%–99% of all translations

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition
  - Pages 194–208