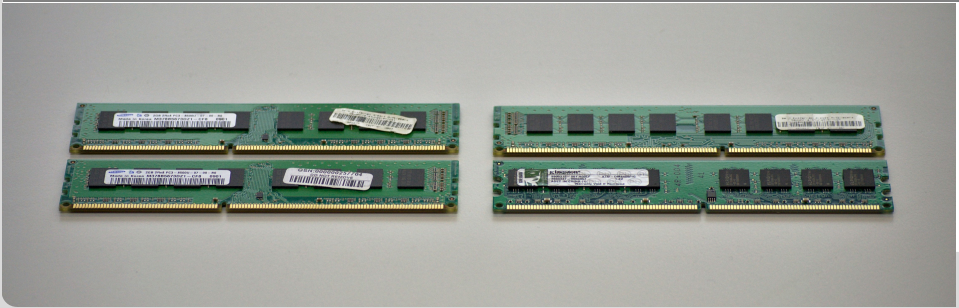
Karlsruhe Institute of Technology

Operating Systems

07. Paging

Prof. Dr. Frank Bellosa | WT 2020/2021

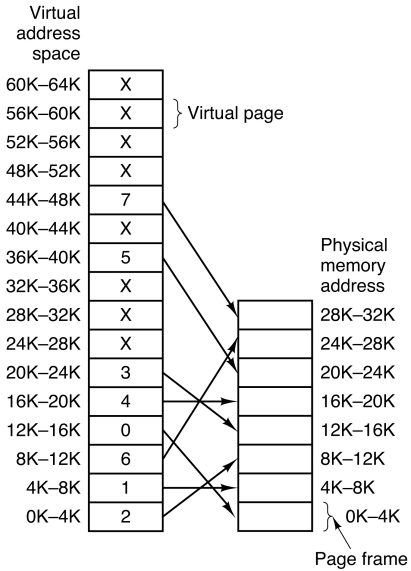


Basic Method

- Divide physical memory into fixed-sized blocks called page frames
  - Size is power of 2 Bytes
  - Typical frame sizes: 4 KiB, 2 MiB, 4 MiB
- Divide virtual memory into blocks called pages
  - Same sizes available as for frames
- OS keeps a page table that stores mappings between virtual page numbers (vpn) and page frame numbers (pfn) for each AS
- OS keeps track of all free frames and modifies page tables as needed
  - To run a program of size n pages, need to find n free frames and load program

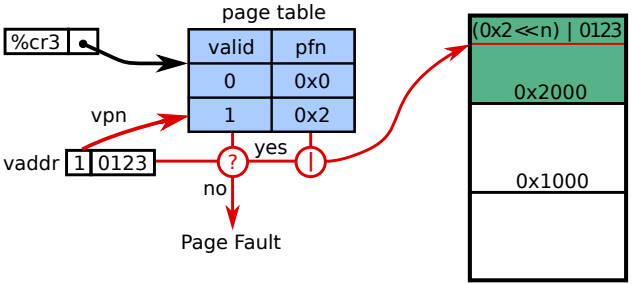
Page Table

- 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)[TB15]



Linear Page Table

- Virtual address is divided into [SGG12]:
  - 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



The OS's Involvement in Paging

The OS performs all operations that require semantic knowledge:

- Page allocation/bringing data into memory
  - The OS needs to find a free page frame for new pages and set up the mapping in the page table of the affected address space
- Page replacement
  - When all page frames are in use, the OS needs to evict pages from memory to make room for new pages
  - e.g., code sections can be dropped and re-read from disk on their next use
  - e.g., heap memory has to be saved to a pagefile or swap area before the frame can be evicted
- Context switching
  - The OS sets the MMU's base register to point to the page table of the next process's address space

Page Table Entry Content

- Valid Bit: Whether the page is currently available in memory or needs to be brought in by the OS, via a page-fault, before accessing it
  (a.k.a. Present Bit)
- Page Frame Number: If the page is present, at which physical address the page is currently located
- Write Bit: If the page may be written to. When a process writes to a page with a clear write bit, the MMU halts the operation and raises a page-fault
- Caching: If this page should be cached at all and with which policy
- Accessed Bit: Set by the MMU if page was touched since the bit was last cleared by the OS
- Dirty Bit: Set by the MMU if this page was modified since the bit was last cleared by the OS

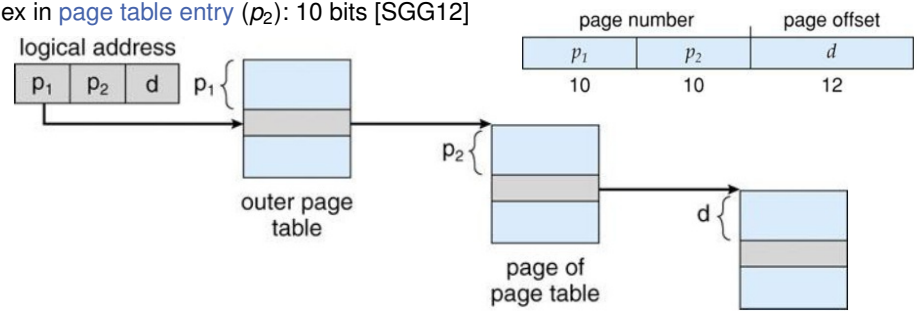
Page Table Structures

Shortcomings of Linear Page Tables

- 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

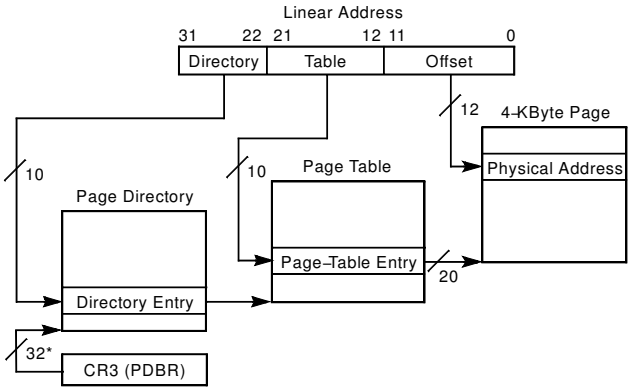
Example: Two-Level Page Table

- 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 [SGG12]



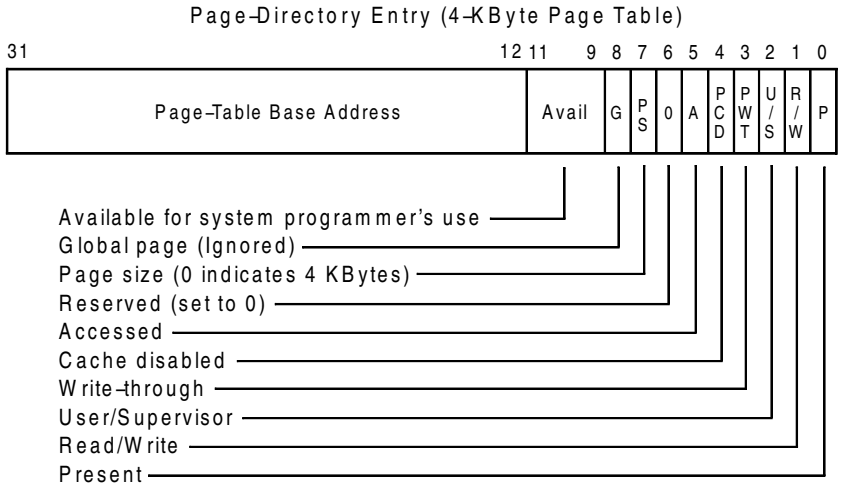
Example: 32 Bit Intel Architecture (IA-32) – Page Table

- Typical page table illustration from the Intel Architecture Manual

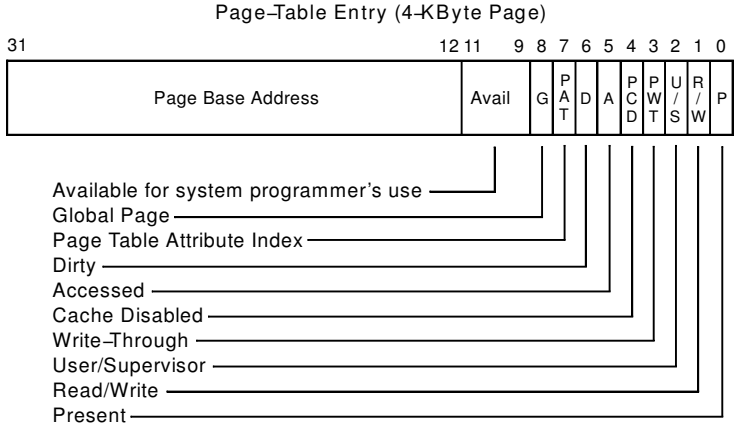


*32 bits aligned onto a 4–KByte boundary

Example: IA-32 – Page Directory



Example: IA-32 – Page Table Entry



Internal Fragmentation

- Paging eliminates external fragmentation due to its fixed size blocks
- With paging, however, internal fragmentation becomes a problem
  - As memory can only be allocated in coarse grained page frame sizes
  - An allocated virtual memory area will generally not end at a page boundary
  - The unused rest of the last allocated page cannot be used by other allocations and is lost

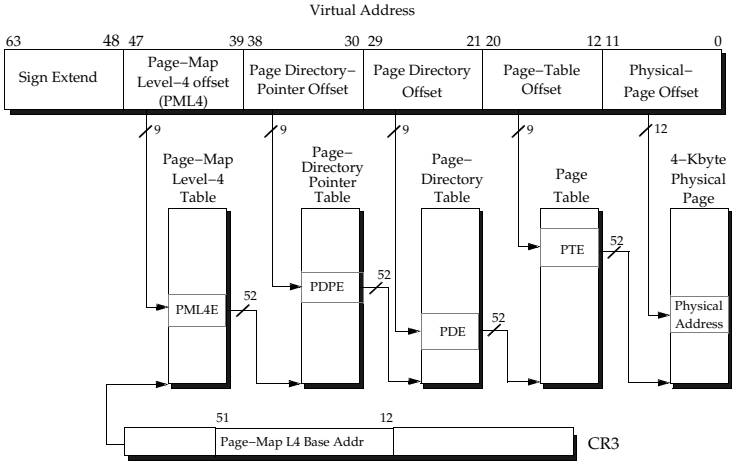
Page Size Trade-Offs

- Fragmentation:
  - Larger pages → More memory wasted due to internal fragmentation for every allocation
  - Small pages → On average only half a page wasted for every allocation
- Table size:
  - Larger pages → Fewer bits needed for pfn (more bits in the offset)
  - Larger pages → Fewer PTEs
  - Smaller pages → More and larger PTEs
  - Note: Page table hierarchies support multiple page sizes with uniform entries, larger pages need fewer page tables (e.g., x86-64)
- I/O:
  - Larger pages → More data needs to be loaded from disk to make page valid
  - Smaller pages → Need to trap to OS more often when loading large program

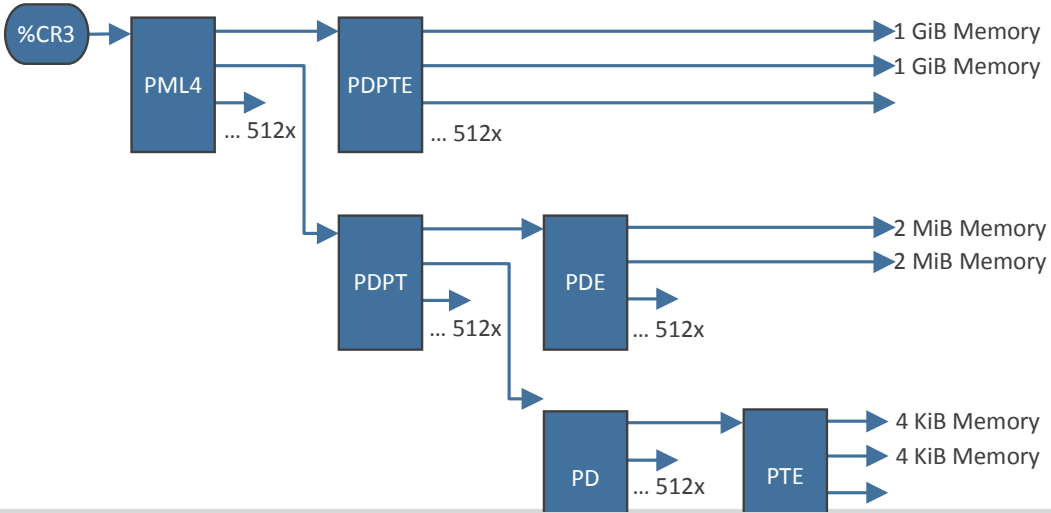
Intel x86-64 Page Table Hierarchy

- x86-64 long mode: 4-level hierarchical page table
- Page directory base register (Control Register 3, %CR3) stores the starting physical address of the first level page table
- For every address space, the page-table hierarchy goes as follows
  - Page map level 4 (PML4)
  - Page directory pointers table (PDPT)
  - Page directory (PD)
  - Page table entry (PTE)
- At each level, the respective table can either point to a directory in the next hierarchy level, or to an entry containing actual mapping data.
- Depending on the depth of the entry, the mapping has different sizes
  - PDPTE: 1 GiB page
  - PDE: 2 MiB page
  - PTE: 4 KiB page

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



Intel x86-64 Page Table Hierarchy



Intel x86-64 Page Table Hierarchy: 5 level paging

- Intel 4-level paging supports a maximum of 256 TiB virtual address space
  - 48 bit linear addresses
  - with 4 KiB page: 9 bit index into PML4, PDPT, PD, PT; 12 bit page offset
- Processors use 46 bit physical addresses (max. 64 TiB physical mem.)

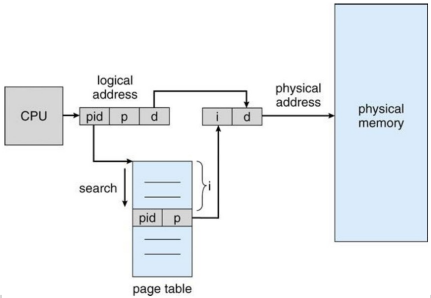
- Intel 5 level pages: extensions for larger address space
  - Add 9 bits for 5th level of hierarchy $\Rightarrow$ 128 PiB virtual memory
  - Physical address width extended up to 52 bit $\Rightarrow$ 4 PiB virtual memory

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 [SGG12]

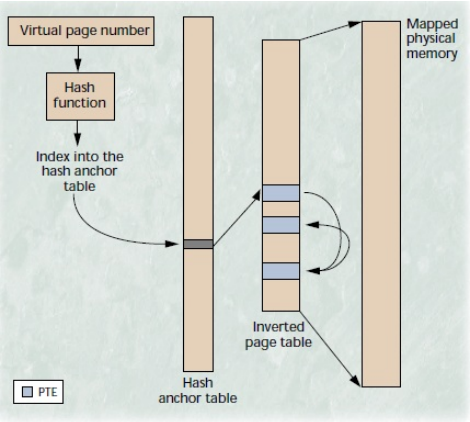
\+ Less overhead for page table meta data

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

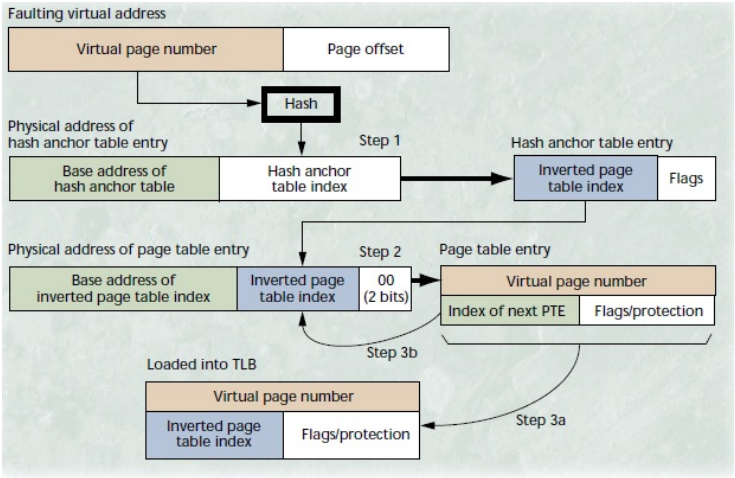


Hashed Inverted Page Table [JM98]

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



Hashed Inverted Page Table Lookup [JM98]



Translation Lookaside Buffer (TLB)

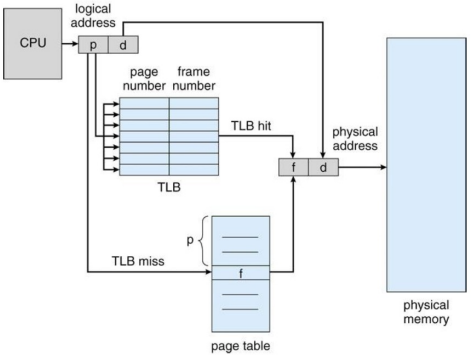
Making Paging Fast

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

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

- **`uint64_t data[512][512];`**
  - Each row is stored in one page (e.g., 4K page size)

Program 1

```
for( j = 0; j < 512; j++ )
   for( i = 0; i < 512; i++ )
      data[i][j] = i*j;
```

$512 \times 512 = 262.144$ TLB misses

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

Program 2

```
for( j = 0; j < 512; j++ )
   for( i = 0; i < 512; i++ )
      data[j][i] = j*i;
```

512 TLB misses

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

References I

[JM98] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.