Address Translation with Paging

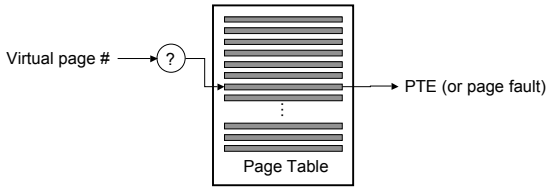
Case studies for X86, SPARC, and PowerPC

Overview

- Page tables
  - What are they? (review)
  - What does a page table entry (PTE) contain?
  - How are page tables organized?
- Making page table access fast
  - Caching entries
  - Translation lookaside buffer (TLB)
  - TLB management

Generic Page Table

- Memory divided into pages
- Page table is a collection of PTEs that maps a virtual page number to a PTE
- Organization and content vary with architecture
- If no virtual to physical mapping => page fault



Generic PTE

- PTE maps virtual page to physical page
- Includes some page properties
  - Valid?, writable?, dirty?, cacheable?

Some acronyms used in this lecture:
- PTE = page table entry
- PDE = page directory entry
- VA = virtual address
- PA = physical address
- VPN = virtual page number
- {R,P}PN = {real, physical} page number

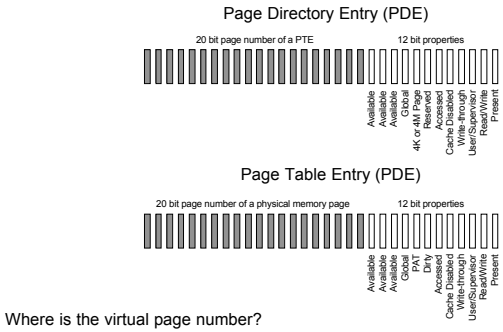
Real Page Tables

- Design requirements
  - Minimize memory use (PT are pure overhead)
  - Fast (logically accessed on every memory ref)
- Requirements lead to
  - Compact data structures
  - O(1) access (e.g. indexed lookup, hashtable)
- Examples: X86 and PowerPC

X86-32 Address Translation

- Page tables organized as a two-level tree
- Efficient because address space is sparse
- Each level of the tree indexed using a piece of the virtual page number for fast lookups
- One set of page tables per process
- Current set of page tables pointed to by CR3
- CPU walks the page tables to find translations
- Accessed and dirty bits updated by CPU
- 4K or 4M (sometimes 2M) pages

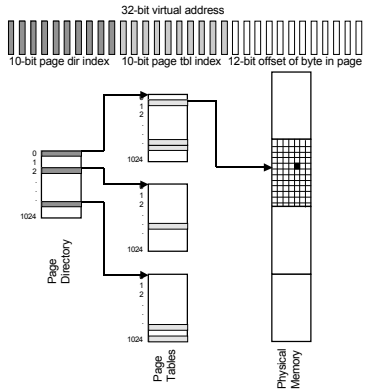
X86-32 PDE and PTE Details



Where is the virtual page number?
If a page is not present, all but bit 0 are available for OS

IA-32 Intel Architecture Software Developer's Manual, Volume 3, pg. 3-24

X86-32 Page Table Lookup



- Top 10 address bits index page directory and return a page directory entry that points to a page table
- Middle 10 bits index the page table that points to a physical memory page
- Bottom 12 bits are an offset to a single byte in the physical page
- Checks made at each step to ensure desired page is available in memory and that the process making the request has sufficient rights to access the page

X86-32 and PAE

- Intel added support for up to 64GB of physical memory in the Pentium Pro - called Physical Address Extensions (PAE)
- Introduced a new CPU mode and another layer in the page tables
- In PAE mode, 32-bit VAs map to 36-bit PAs
- Single-process address space is still 32 bits
- 4-entry page-directory-pointer-table (PDPT) points to a page directory and then translation proceeds as normal
- Page directory and page table entries expanded to 64 bits to hold 36 bit physical addresses
- Only 512 entries per 4K page
- 4K or 2M page sizes

What about 64-bit X86?

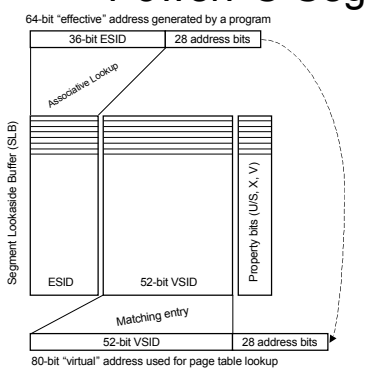
- X86-64 (AMD64 or EM64T) supports a 64-bit virtual address (only 48 bits effective)
- Three modes
  - Legacy 32-bit (32-bit VA, 32-bit PA)
  - Legacy PAE (32-bit VA, up to ***52***-bit PA)
  - Long PAE mode (64-bit VA, 52-bit PA)
- Long mode requires ***four*** levels of page tables to map 48-bit VA to 52-bit PA

AMD64 Architecture Programmer's Manual Volume 2: System Programming, Ch. 5

PowerPC Address Translation

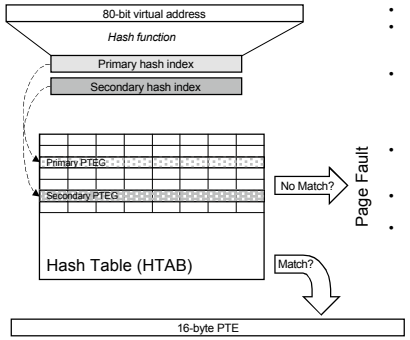
- 80-bit virtual address obtained via PowerPC segmentation mechanism
- 62-bit physical ("real") address
- PTEs organized in a hash table (HTAB)
- Each HTAB entry is a page table entry group (PTEG)
- Each PTEG has (8) 16-byte PTEs
- Hash function on VPN gives the index of ***two*** PTEGs (Primary and secondary PTEGs)
- Resulting 16 PTEs searched for a VPN match
- No match => page fault

PowerPC Segmentation



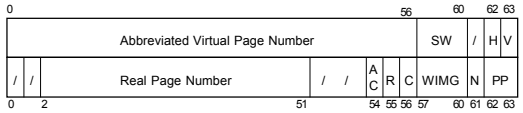
- SLB is an "associative memory"
- Top 36 bits of a program-generated "effective address used as a tag called the effective segment id (ESID)
- Search for tag value in SLB
- If a match exists, property bits validated for access
- A failed match causes segment fault
- Associated 52-bit virtual segment id (VSID) is concatenated with the remaining address bits to form an 80-bit virtual address
- Segmentation used to separate processes within the large virtual address space

PowerPC Page Table Lookup



- Variable size hash table
- Processor register points to hash table base and gives table's size
- Architecture-defined hash function on virtual address returns two possible hash table entries
- Each of the 16 possible PTEs is checked for a VA match
- If no match then page fault
- Possibility that a translation exists but that it can't fit in the hash table – OS must handle

PowerPC PTE Details



Key
SW=Available for OS use
H=Hash function ID
V=Valid bit
AC=Address compare bit
R=Referenced bit
C=Changed bit
WIMG=Storage control bits
N=No execute bit
PP=Page protection bits

- 16-byte PTE
- Both VPN and RPN
- Why only 57 bit VPN?

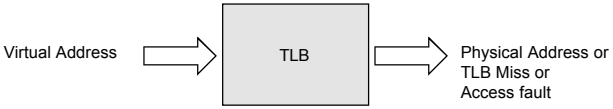
PowerPC Operating Environment Architecture, Book III, Version 2.01, Sections 4.3-4.5

Making Translation Fast

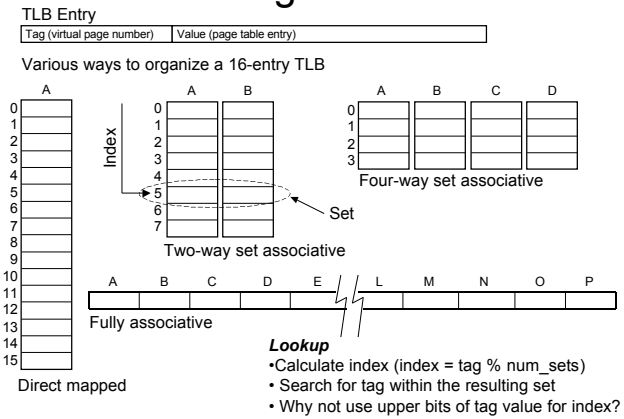
- Page table logically accessed on *every instruction*
- Paging has turned each memory reference into at least three memory references
- Page table access has temporal locality
- Use a cache to speed up access
- Translation Lookaside Buffer (TLB)

Generic TLB

- Cache of recently used PTEs
- Small – usually about 64 entries
- Huge impact on performance
- Various organizations, search strategies, and levels of OS involvement possible
- Consider X86 and SPARC



TLB Organization



Lookup

- Calculate index (index = tag % num_sets)
- Search for tag within the resulting set
- Why not use upper bits of tag value for index?

Associativity Trade-offs

- Higher associativity
  - Better utilization, fewer collisions
  - Slower
  - More hardware
- Lower associativity
  - Fast
  - Simple, less hardware
  - Greater chance of collisions
- How does page size affect TLB performance?

X86 TLB

- TLB management shared by processor and OS
- CPU fills TLB on demand from page table (the OS is unaware of TLB misses)
- CPU evicts entries when a new entry must be added and no free slots exist
- Operating system ensures TLB/page table consistency by flushing entries as needed when the page tables are updated or switched (e.g. during a context switch)
- TLB entries can be removed by the OS one at a time using the INVLPG instruction or the entire TLB can be flushed at once by writing a new entry into CR3

Example: Pentium-M TLBs

- Four different TLBs
  - Instruction TLB for 4K pages
    - 128 entries, 4-way set associative
  - Instruction TLB for large pages
    - 2 entries, fully associative
  - Data TLB for 4K pages
    - 128 entries, 4-way set associative
  - Data TLB for large pages
    - 8 entries, 4-way set associative
- All TLBs use LRU replacement policy
- Why different TLBs for instruction, data, and page sizes?

SPARC TLB

- SPARC is RISC (simpler is better) CPU
- Example of a “software-managed” TLB
- TLB miss causes a fault, handled by OS
- OS explicitly adds entries to TLB
- OS is free to organize its page tables in any way it wants because the CPU does not use them
- E.g. Linux uses a tree like X86, Solaris uses a hash table

Minimizing Flushes

- On SPARC, TLB misses trap to OS (SLOW)
- We want to avoid TLB misses
- Retain TLB contents across context switch
- SPARC TLB entries enhanced with a *context id*
- Context id allows entries with the same VPN to coexist in the TLB (e.g. entries from different process address spaces)
- When a process is switched back onto a processor, chances are that some of its TLB state has been retained from the last time it ran
- Some TLB entries shared (OS kernel memory)
  - Mark as global
  - Context id ignored during matching

Example:UltraSPARC III TLBs

- Five different TLBs
- Instruction TLBs
  - 16 entries, fully associative (supports all page sizes)
  - 128 entries, 2-way set associative (8K pages only)
- Data TLBs
  - 16 entries, fully associative (supports all page sizes)
  - 2 x 512 entries, 2-way set associative (each supports one page size per process)
- Valid page sizes – 8K (default), 64K, 512K, and 4M
- 13-bit context id – 8192 different concurrent address spaces
- What happens if you have > 8192 processes?

Speeding Up TLB Miss Handling

- In some cases a huge amount of time can be spent handling TLB misses (2-50% in one study of SuperSPARC and SunOS)
- Many architectures that use software managed TLBs have hardware assisted TLB miss handling
- SPARC uses a large, virtually-indexed, direct-mapped, physically contiguous table of recently used TLB entries called the Translation Storage Buffer (TSB)
- The location of the TSB is loaded into the processor on context switch (implies one TSB per process)
- On TLB miss, hardware calculates the offset of the matching entry into the TSB and supplies it to the software TLB miss handler
- In most cases, the software TLB miss handler only needs to make a tag comparison to the TSB entry, load it into the TLB, and return
- If an access misses in the TSB then a slow software search of page tables is required