UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Address Translation Case Studies

Questions answered in this lecture:

- Case studies: x86, PowerPC, SPARC architecture
- How are page tables organized?
- What do PTEs contain?
- How can address translation be performed quickly?

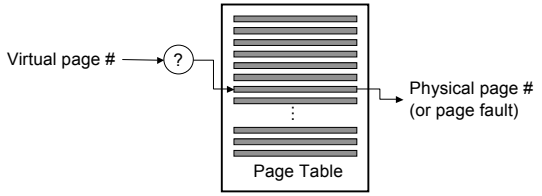
Generic Page Table

Memory divided into pages

Page table is a collection of PTEs (page table entries)

- Maps a virtual page number to physical page (or another page table)
- If no virtual to physical mapping => seg fault or page fault

Organization and content vary with architecture



Generic PTE

PTE maps virtual page to physical page

Includes some page properties

- Valid?, writable?, dirty?, cacheable?

Virtual Page #	Physical Page #	Property bits

Terminology
- 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

Designing Page Tables

Requirements/Goals

- Minimize memory use (PTs are pure overhead)
- Fast (logically accessed on every memory ref)

Requirements lead to

- Compact data structures
  - Support sparse address spaces
- O(1) access
  - e.g. indexed lookup, hashtable

Case Studies: x86 and PowerPC

Case Study 1: x86-32

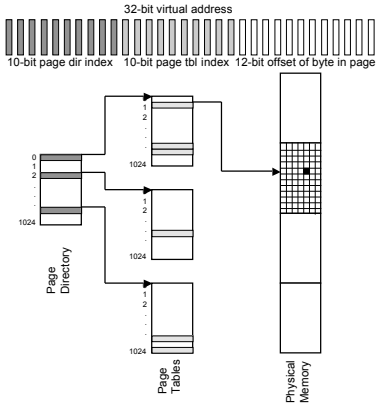
Basic Idea: Page tables organized as a two-level tree
- "Outer" and "inner" page tables
- Efficient because address space is sparse
- Each level of tree indexed using part of VPN for fast lookups

Hardware support
- CPU walks the page tables to find translations
- Architecturally-defined page tables
- CPU updates accessed and dirty bits (useful for policies in future lectures)

How does CPU know where page tables live?
- Current set of page tables pointed to by `CR3`
- One set of page tables per process

Page sizes: 4K or 4M (sometimes 2M)

x86-32 Page Table Lookup



Top 10 va bits:
- index page directory
- returns PDE that points to a page table

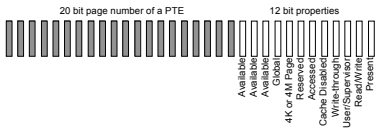
Next 10 va bits:
- index page table
- returns PTE that points to a physical memory page

Bottom 12 va bits:
- offset to a single byte in the physical page

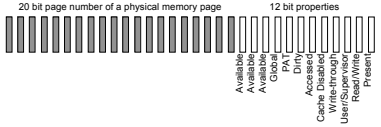
Checks made at each step:
- Ensure desired page is available in memory
- Process has sufficient rights to access page

x86-32 PDE and PTE Details

Page Directory Entry (PDE)



Page Table Entry (PTE)



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 and PAE

Problem:
- How to handle larger physical memory sizes??
- How much physical memory can 32 bits address?

Pentium Pro adds support for up to 64 GB of physical memory
- How many physical address bits are needed?

Idea:
- Map 32-bit address space to 36-bit physical space
- Single-process logical address space is still 32 bits

Physical Address Extensions (PAE): Two additions
- New CPU mode: PAE
  - In PAE mode, 32-bit VAs map to 36-bit PAs
- Another layer in the page tables needed
  - To hold 24-bit PPN, PDEs and PTEs expanded to 64 bits
    - How many PTE fit in a 4KB page now?
  - Still want to be able to fit page tables in a page
  - Top-level: 4-entry `page-directory-pointer-table (PDPT)`
    - Points to a page directory and then translation proceeds as before

Case Study #2: PowerPC

Goal: Handle large virtual address spaces w/o large page tables

- 80-bit virtual address with segments
- 62-bit physical ("real") address

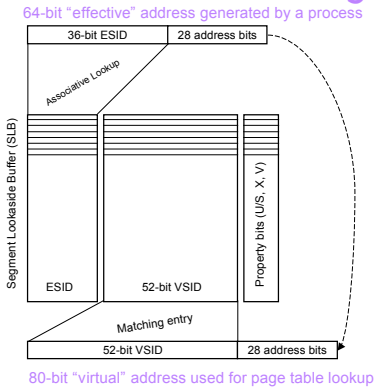
Idea: Inverted Page Tables

- Only need entries for virtual pages w/ physical mappings
- Too much time to search entire table
- Find relevant PTE with hash function

Details of Hash Table (HTAB)

- Hash table (HTAB) contains PTEs
- Each HTAB entry is a page table entry group (PTEG)
- Each PTEG has 8 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
- Why is it good to have PTEGs?

PowerPC Segmentation



Segment Lookaside Buffer (SLB) is an "associative memory"
- Small # entries

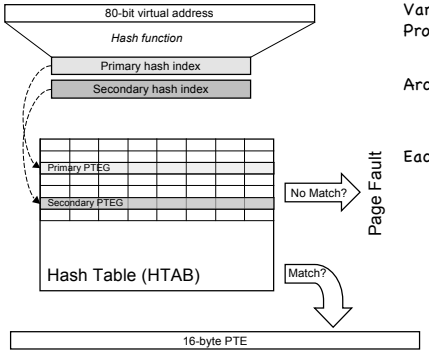
Top 36 bits of a process "effective" address used as "effective segment id (ESID)" tag

Search for ESID tag value in SLB
- If a match exists, property bits validated for access
- A failed match causes fault

Associated 52-bit virtual segment id (VSID) is concatenated with the remaining 28 bits to form an 80-bit virtual address

Segmentation used to separate processes within the large virtual address space

PowerPC Page Table Lookup



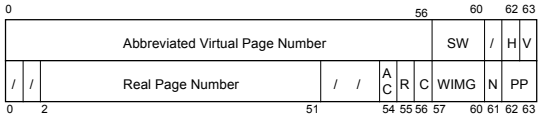
Variable size hash table

Processor register points to hash table base and bounds

Architecture-defined hash function on VPN 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

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

TLBs: Making Translation Fast

Problem:

- Page table logically accessed on *every instruction*
- Each process memory reference requires at least three memory references

Observation:

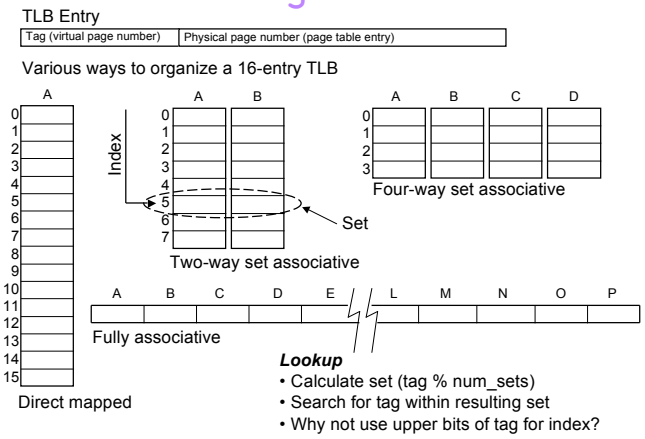
- Page table access has temporal locality
- Use a cache to speed up access

Translation Lookaside Buffer (TLB)

- Small cache of recent PTEs (about 64 entries)
- Huge impact on performance

Various organizations, search strategies, and levels of OS involvement possible

TLB Organization



Lookup

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

TLB 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?
  - TLB reach or coverage

Case Study: x86 TLB

TLB management

- Shared by processor and OS

Role of CPU

- Fills TLB on demand from page table (the OS is unaware of TLB misses)
- Evicts entries when a new entry is added and no free slots exist

Role of OS

- Ensures TLB/page table consistency by flushing entries as needed when page tables are updated or switched (e.g. during context switch)
- TLB entries removed one at a time using INVLPG instruction
- Or entire TLB flushed 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 and data?

Case Study: SPARC TLB

SPARC is RISC CPU
- Philosophy of "simpler is better"

TLB management
- "Software-managed" TLB
- TLB miss causes a fault, handled by OS
- Page table format not defined by architecture!

Role of OS
- Explicitly adds entries to TLB
- Free to organize page tables in any way it wants because the CPU does not use them
  - E.g. Linux uses multiple levels like x86, Solaris uses a hash table

Speedup SPARC TLB: #1 Minimizing TLB Flushes

Problem: TLB misses trap to OS (SLOW)
- We want to avoid TLB misses

Idea:
- Retain TLB contents across context switch
- When a process is switched back onto CPU, chances are that some of its TLB entries have been retained from last time

SPARC TLB entries enhanced with a *context id*
- Context id allows entries with same VPN to coexist in the TLB (e.g. entries from different process address spaces)
- Some TLB entries shared (OS kernel memory)
  - Mark as global
  - Context id ignored during matching

Example: UltraSPARC III TLBs

Supports multiple page sizes
- 8K (default), 64K, 512K, and 4M

Five different TLBs

2 Instruction TLBs
- 16 entries, fully associative (supports all page sizes)
- 128 entries, 2-way set associative (8K pages only)

3 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)

13-bit context id
- 8192 different concurrent address spaces
- What happens if you have > 8192 processes?

Speedup SPARC TLB: #2 Faster TLB Miss Handling

Problem with software-managed TLBs:

- Huge amount of time can be spent handling TLB misses (2-50% in one study of SuperSPARC and SunOS)

Idea:

- Have hardware-assisted TLB miss handling

SPARC solution:

- Use large, virtually-indexed, direct-mapped, physically contiguous table of recently used TLB entries
- Translation Storage Buffer (TSB)
- On TLB miss, hardware calculates offset of matching entry in TSB and supplies it to software TLB miss handle
  - 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 TSB then a slow software search of page tables is required
- Context switch: Location of TSB is loaded into the processor (implies one TSB per process)