CS162
Operating Systems and Systems Programming
Lecture 13

Address Translation (con't)
Caches and TLBs

March 2, 2010

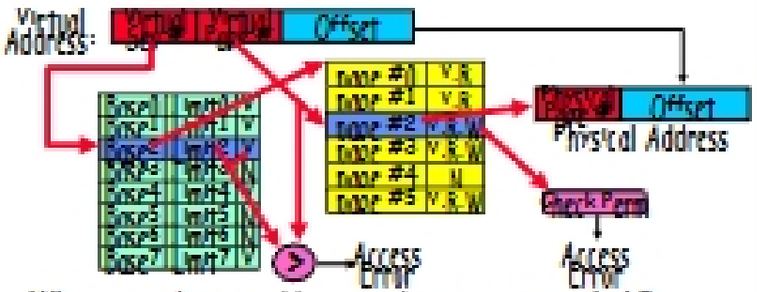
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Review: Single-Level Translation

- Advantages
  - Low translation overhead
  - Simplicity
- Disadvantages
  - Large page tables
    - E.g., 32b address space, 4KB pages → up to 2^{20} = 1mil page entries for each process
  - Expensive to share memory
    - E.g., 4KB pages, want to share 100MB → need to update 25,000 entries in page table

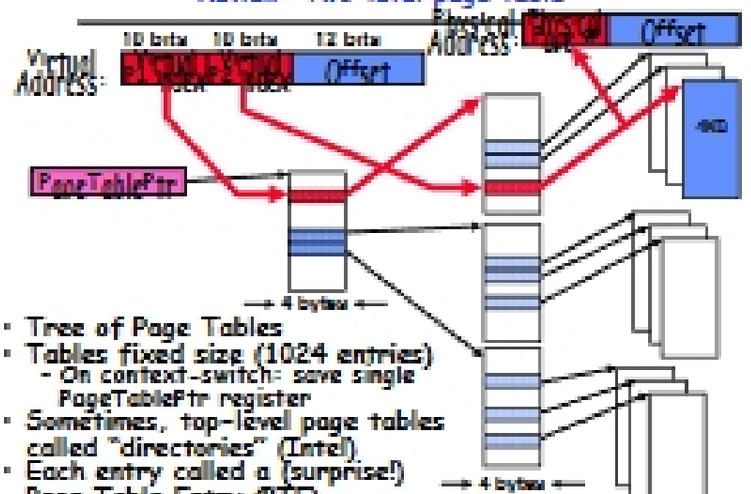
Review: Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

Review: Two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Sometimes, top-level page tables called "directories" (Intel)
- Each entry called a (surprise!) Page Table Entry (PTE)

Goals for Today

- Finish discussion of both Address Translation and Protection
- Caching and TLBs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - Region of address space is actually invalid or
    - Page/directory is just somewhere else than memory
  - Validity checked first
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives copy of parent address space to child
    - Address spaces disconnected after child created
  - How to do this cheaply?
    - Make copy of parent's page tables (point at same memory)
    - Mark entries in both sets of page tables as read-only
    - Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

How is the translation accomplished?

- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
  - For each virtual address, takes page table base pointer and traverses the page table in hardware
  - Generates a "Page Fault" if it encounters invalid PTE
    - Fault handler will decide what to do
    - More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!
- In fact, need way to cache translations for either case!

Dual-Mode Operation

- Can Application modify its own translation tables?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Priviledge Level) from 0 - 3
    - PL0 has full access. PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Typical OS kernels on Intel processors only use PL0 ("user") and PL3 ("kernel")

For Protection, Lock User-Programs in Asylum

- Idea: Lock user programs in padded cell with no exit or sharp objects
  - Cannot change mode to kernel mode
  - User cannot modify page table mapping
  - Limited access to memory: cannot adversely affect other processes
    - Side-effect: Limited access to memory-mapped I/O operations
    - I/O that occurs by read/write memory locations
  - Limited access to interrupt controller

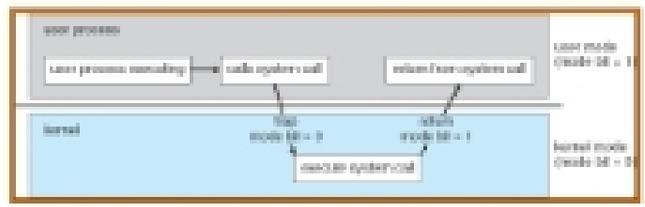
- A couple of issues
  - How to share CPU between kernel and user programs?
    - Kinda like both the inmates and the warden in asylum are the same person! How do you manage this?
  - How do programs interact?
  - How does one switch between kernel and user modes?
    - OS → user (kernel → user mode): getting into cell
    - User → OS (user → kernel mode): getting out of cell

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - Point at code in memory so program can execute
    - Possibly point at statically initialized data
  - Run Program:
    - Set machine registers
    - Set hardware pointer to translation table
    - Set processor status word for user mode
    - Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - No! Only specific ones.
  - System call ID encoded into system call instruction
    - Index forces well-defined interface with kernel