CS 3204
Operating Systems

Lecture 17

Godmar Back

Virginia Tech

Announcements

- Project 2 due tonight
  - Reminder: by end of semester, passing students will have provided a deliverable that achieves >= 90% test score on project 2 (or a 100% test score on project 3 or 4's regression tests.)
- Additional office hours
  - Xiaomo: today 3:00 - 5:00pm
- Project 3 Help Sessions
  - Thursday 22nd 7-9pm McB 216
  - Friday 23rd 5-7pm McB 216
- Project 3 Design Milestone
  - Monday 26th 11:59pm – no extensions!
- Midterm March 29
  - See announcement + sample midterms on class website

Implementing Page Tables

- Many, many variations possible
- Done in combination of hardware & software
  - Hardware part: dictated by architecture
  - Software part: up to OS designer
    - Machine-dependent layer that implements architectural constraints (what hardware expects)
    - Machine-independent layer that manages page tables
- Must understand how TLB works first

Page Tables Function & TLB

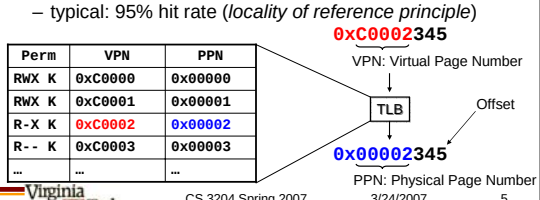
Trans (with paging):
{ Process Ids } × { Virtual Addresses } × { user, kernel } × { read, write, execute }

→ { Physical Addresses } ∪ { INVALID } ∪ { Some Location On Disk }

- For each combination (process id, virtual_addr, mode, type of access) must decide
  - If access is permitted
  - If permitted:
    - if page is resident, use physical address
    - if page is non-resident, page table has information on how to get the page in memory
- CPU uses TLB for actual translation – page table feeds the TLB on a TLB miss

TLB: Translation Look-Aside Buffer

- Virtual-to-physical translation is part of every instruction (why not only load/store instructions?)
  - Thus must execute at CPU pipeline speed
- TLB caches a number of translations in fast, fully-associative memory
  - typical: 95% hit rate (*locality of reference principle*)

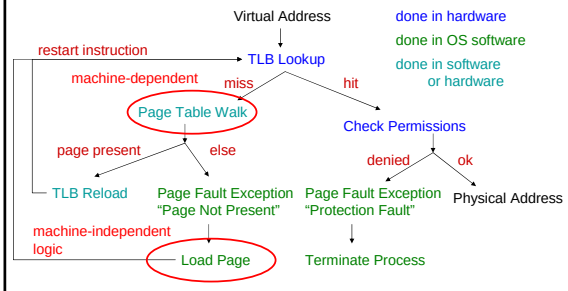


Perm	VPN	PPN
RWX K	0xC0000	0x00000
RWX K	0xC0001	0x00001
R-X K	0xC0002	0x00002
R-- K	0xC0003	0x00003
…	…	…

TLB Management

- Note: on previous slide example, TLB entries did not have a process id
  - As is true for x86
- Then: if process changes, some or all TLB entries may become invalid
  - X86: flush entire TLB on process switch (refilling adds to cost!)
- Some architectures store process id in TLB entry (MIPS)
  - Flushing (some) entries only necessary when process id reused

Address Translation & TLB



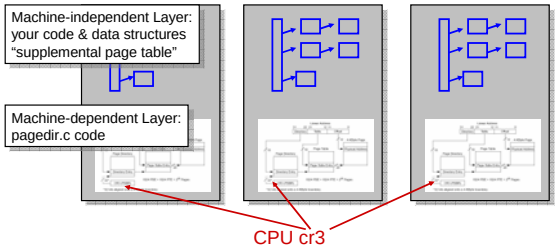
TLB Reloaded

- TLB small: typically only caches 64-2,048 entries
  - What happens on a miss? – must consult ("walk") page table – TLB Reload or Refill
- TLB Reload in software (MIPS)
  - Via TLB miss handlers – OS designer can pick any page table layout – page table is only read & written by OS
- TLB Reload in hardware (x86)
  - Hardware & software must agree on page table layout *inasmuch* as TLB miss handling is concerned – page table is read by CPU, written by OS
- Some architectures allow either (PPC)

Page Tables vs TLB Consistency

- No matter which method is used, OS must ensure that TLB & page tables are consistent
  - On multiprocessor, this may require "TLB shootdown"
- For software-reloaded TLB: relatively easy
  - TLB will only contain what OS handlers place into it
- For hardware-reloaded TLB: two choices
  - Use same data structures for page table walk & page loading (hardware designers reserved bits for OS's use in page table)
  - Use a layer on top (facilitates machine-independent implementation) – this is the recommended approach for Pintos Project 3
    - In this case, must update actual page table (on x86: "page directory") that is consulted by MMU during page table walk
    - Code is already written for you in pagedir.c

Hardware/Software Split in Pintos



Representing Page Tables

- Choice impacts speed of access vs size needed to store mapping information:
  - Simple arrays (PDP-11, VAX)
    - Fast, but required space makes it infeasible for large, non-continuous address spaces
  - Search trees (aka "hierarchical" or "multi-level" page tables)
  - Hash table

Two-level Page Table

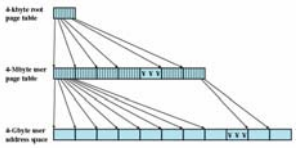
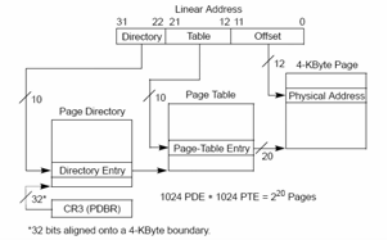


Figure 8.4 A Two-Level Hierarchical Page Table

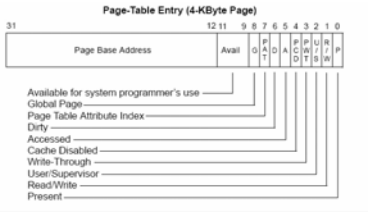
- Q.: how many pages are needed in
  - Minimum case
  - Worst case? (what is the worst case?)

Example: x86 Address Translation



- Two-level page table
- Source: [IA32-v3] 3.7.1

Example: x86 Page Table Entry

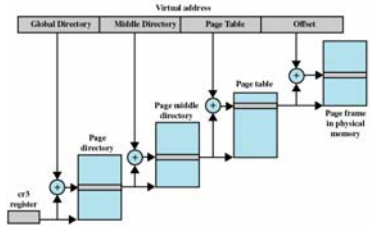


- Note: if bit 0 is 0 ("page not present") MMU will ignore bits 1-31 – OS can use those at will

Page Table Management on Linux

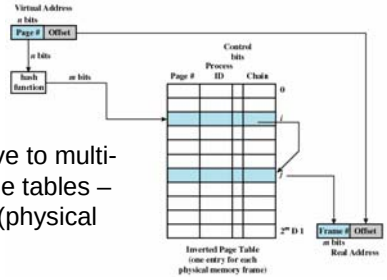
- Interesting history:
  - Linux was originally x86 only with 32bit physical addresses. Its page table matched the one used by x86 hardware
  - Since:
    - Linux has been ported to other architectures
    - x86 has grown to support 36bit physical addresses (PAE) – required 3-level page table
- Linux's now uses 4-level page table to support 64-bit architectures

Linux Page Tables (2)



- On x86 – hardware == software
  - On 32-bit (no PAE) middle directory disappears
- With four-level, "PUD" page upper directory is added (not shown)

Inverted Page Tables



- Alternative to multi-level page tables – size is O(physical memory)

Summary

- Page tables store mapping information from virtual to physical addresses, or to find non-resident pages
  - Input is: process id, current mode (user/kernel) and kind of access (read/write/execute)
- TLBs cache such mappings
- Page tables are consulted when TLB miss occurs
  - Either all software, or in hardware
- OS must maintain its page table(s) and, if hardware TLB reload is used, the page table (on x86 aka "page directory + table") that is consulted by MMU
  - These two tables may or may not be one and the same
- The OS page table must have sufficient information to load a page's content from disk