CS 3204
Operating Systems

Lecture 17

Godmar Back

Announcements

- Project 3 milestone due October 26
  - No extensions here, accept early submissions (send us email for prompt grading!)
- Project 3 due November 8
  - If your project 2 still doesn't 100% work, seek help now
- Project 3 Help Sessions
  - Thursday (tonight) McB 126 7-9pm
  - Monday McB 113 7-9pm
- Midterm graded
- Reading assignment:
  - Chapter 8 & 9

Virtual Memory

Page Tables & TLB (cont'd)

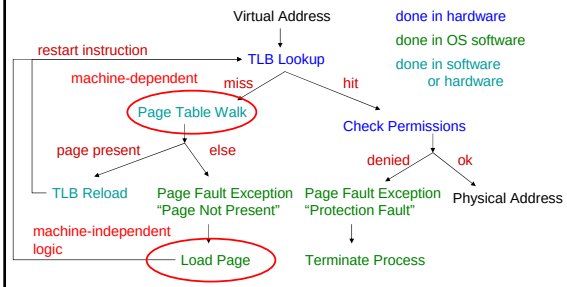
Page Tables Function & TLB

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

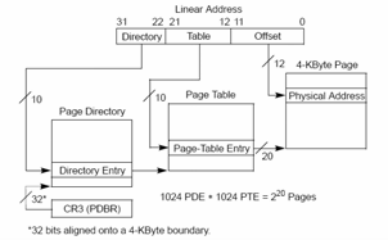
Address Translation & TLB



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

Example: x86 Address Translation



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

Two-level Page Table

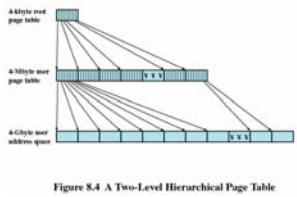
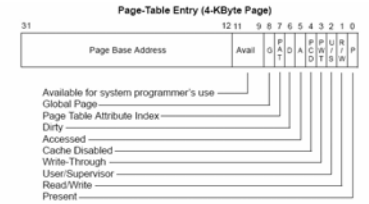


Figure 8.4 A Two-Level Hierarchical Page Table

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

Example: x86 Page Table Entry

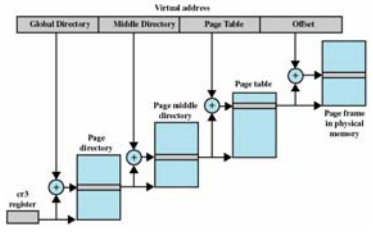


- Note: if bit 0 is 0 ("page not present") MMU will ignore bits 1-31 – OS can use those at will

Page Table Management on Linux

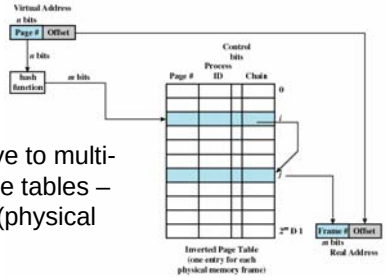
- Interesting history:
  - Linux was originally x86 only with 32bit physical addresses. Its page table matched the one used by x86 hardware
  - Since:
    - Linux has been ported to other architectures
    - x86 has grown to support 36bit physical addresses (PAE) – required 3-level page table
- Linux's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

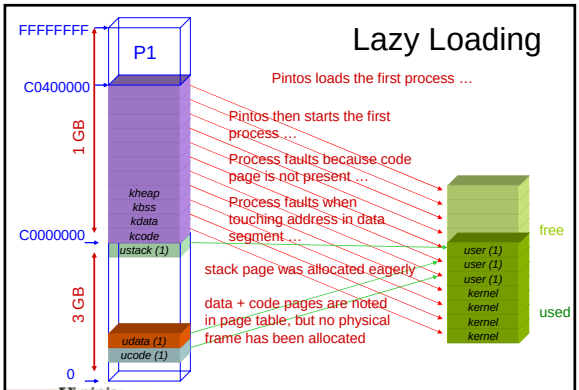
Virtual Memory

Paging Techniques

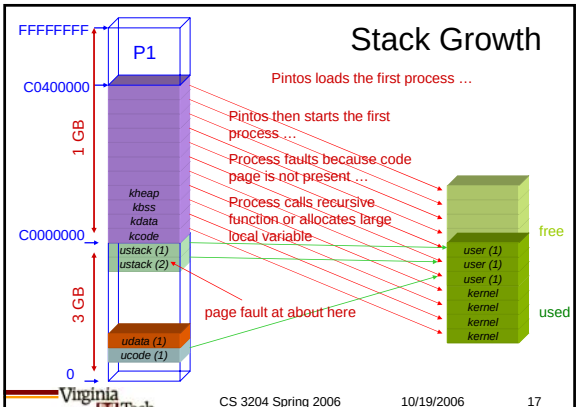
Demand paging

- Idea: only keep data in memory that's being used
  - Needed for virtualization – don't use up physical memory for data processes don't access
- Requires that actual allocation of physical page frames be delayed until first access
- Many variations
  - Lazy loading of text & data, mmapped pages & newly allocated heap pages
  - Copy-on-write

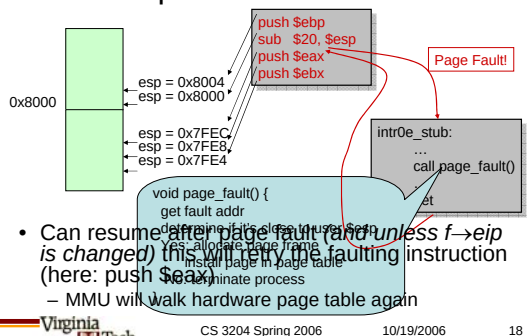
Lazy Loading



Stack Growth



Microscopic View of Stack Growth



- Can resume after page fault (*and unless f→eip is changed*) this will retry the faulting instruction (here: push $eax)
  - MMU will walk hardware page table again

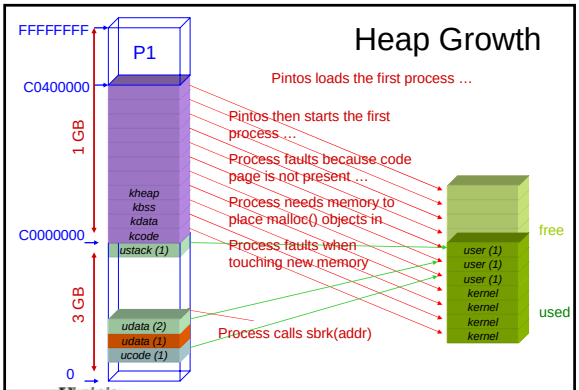
Fault Resumption

- Requires that faulting CPU instruction be restartable
  - Most CPUs are designed this way
- Very powerful technique
  - Entirely transparent to user program: user program is frozen in time until OS decides what to do
- Can be used to emulate lots of things
  - Programs that just ignore segmentation violations (!?) (here: resume with next instruction – retrying would fault again)
  - Subpage protection (protect entire page, take fault on access, check if address was to an valid subpage region)
  - Virtual machines (vmware, qemu – run entire OS on top of another OS)
  - Garbage collection
  - Distributed Shared Memory

Distributed Shared Memory

- Idea: allows accessing other machine's memory as if it were local
- Augment page table to be able to keep track of network locations:
  - local virtual address → (remote machine, remote address)
- On page fault, send request for data to owning machine, receive data, allocate & write to local page, map local page, and resume
  - Process will be able to just use pointers to access all memory distributed across machines – fully transparent
- Q.: how do you guarantee consistency?
  - Lots of options

Heap Growth



mmap()