CS 3204 Operating Systems

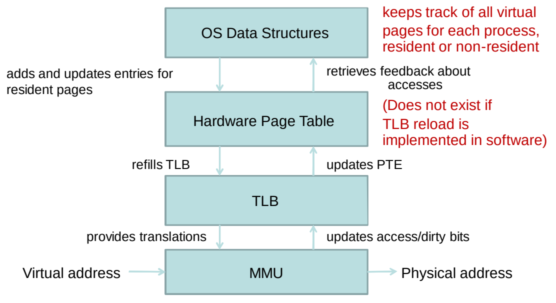
Lecture 16

Godmar Back

Announcements

- Project 3 Milestone due Friday Oct 24, 11:59pm
  - No extensions
- Will return by Monday
- Project 3 Help Session next Monday 6-8pm
  - Room TBA – check forum
- Read book chapters 8 and 9 on memory management
- Reminder: need to pass 90% of tests of project 2 by the end of the semester to pass the class
  - All project 2 tests except multi-oom will appear as regression tests in project 3 and 4

Layers in Address Translation


OS Data Structures
keeps track of all virtual pages for each process, resident or non-resident
adds and updates entries for resident pages
retrieves feedback about accesses
Hardware Page Table
(Does not exist if TLB reload is implemented in software)
refills TLB
updates PTE
TLB
provides translations
updates access/dirty bits
Virtual address
MMU
Physical address

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

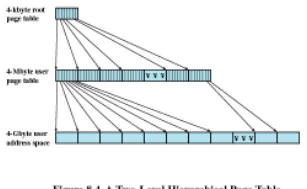
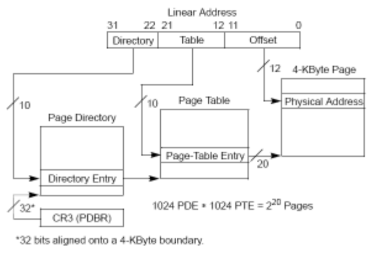


Figure 8.4 A Two-Level Hierarchical Page Table

- This is the original page design on i386
  - Next slide provides details on how address bits index tree levels
- If address space is sparse, interior nodes/paths do not exist (parent contains a NULL)
- Picture shown shows how tree divides virtual address space
- The kernel virtual pages that hold the page table information need not be contiguous

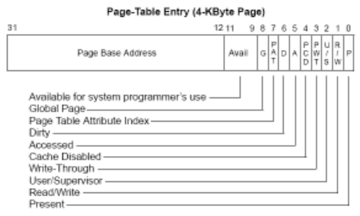
- Q.: how many pages are needed in
  - Minimum case
  - Worst case? (what is the worst case?)

Example: x86 Address Translation



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

Example: x86 Page Table Entry

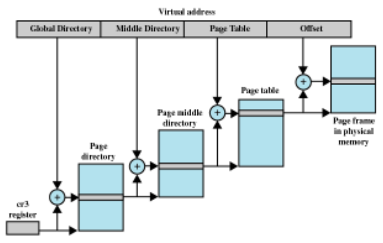


- Note: if bit 0 is 0 ("page not present") MMU will ignore bits 1-31 – OS can use those at will

Page Table Management on Linux

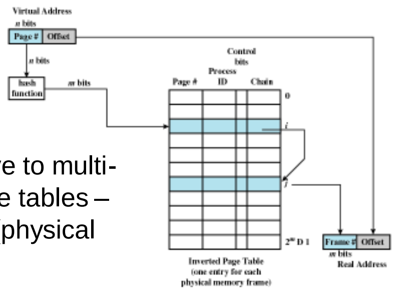
- Interesting history:
  - Linux was originally x86 only with 32bit physical addresses. Its page table matched the one used by x86 hardware
  - Since:
    - Linux has been ported to other architectures
    - x86 has grown to support 36bit physical addresses (PAE) – required 3-level page table
- Linux's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

Page Tables - 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

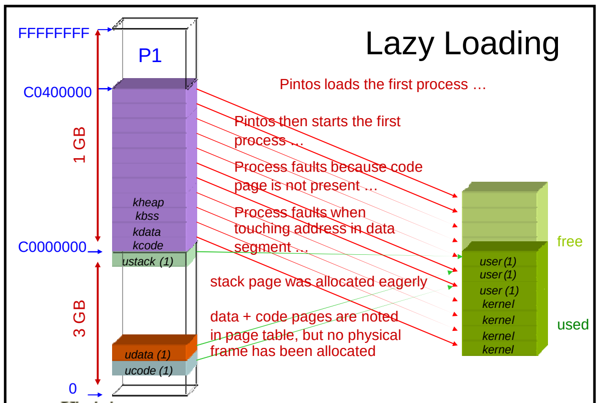
Virtual Memory

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



Pintos loads the first process …

Pintos then starts the first process …

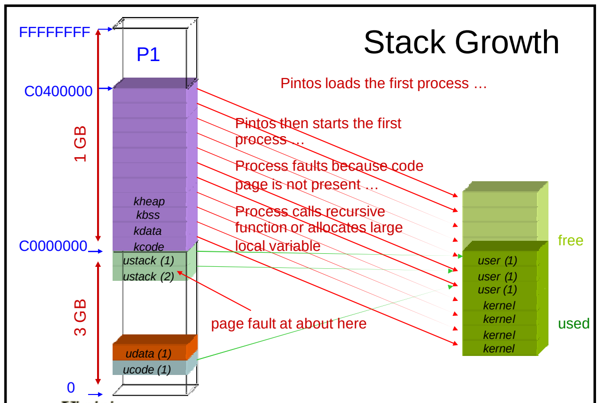
Process faults because code page is not present …

Process faults when touching address in data segment …

stack page was allocated eagerly

data + code pages are noted in page table, but no physical frame has been allocated

Stack Growth



Pintos loads the first process …

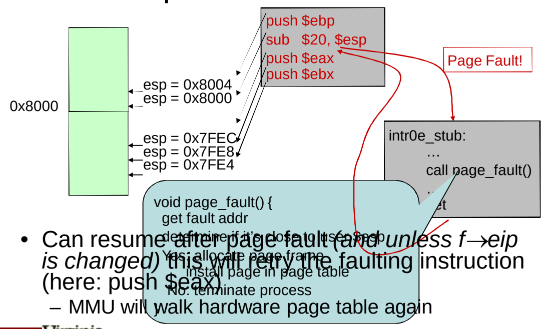
Pintos then starts the first process …

Process faults because code page is not present …

Process calls recursive function or allocates large local variable

page fault at about here

Microscopic View of Stack Growth



```
push $ebp
sub   $20, $esp
push $eax
push $ebx
```

```
void page_fault() {
  get fault addr
  determine if it's close to user $esp
  Yes: allocate page frame
     install page in page table
  No: terminate process
```

- 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