CS 537
Lecture 8
Paging

Michael Swift

Paging Advantages

- Easy to allocate physical memory
  - physical memory is allocated from free list of frames
    - to allocate a frame, just remove it from its free list
  - external fragmentation is not a problem!
    - complication for kernel contiguous physical memory allocation
      - many lists, each keeps track of free regions of particular size
      - regions' sizes are multiples of page sizes
      - "buddy algorithm"
- Easy to "page out" chunks of programs
  - all chunks are the same size (page size)
  - use valid bit to detect references to "paged-out" pages
  - also, page sizes are usually chosen to be convenient multiples of disk block sizes

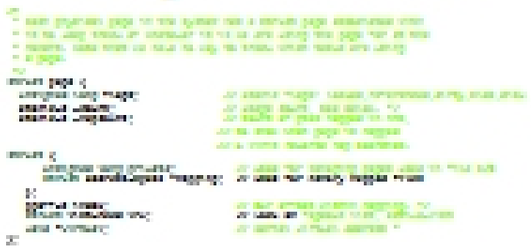
Paging Disadvantages

- Can still have internal fragmentation
  - process may not use memory in exact multiples of pages
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - solution: use a hardware cache to absorb page table lookups
    - translation lookaside buffer (TLB)
- Memory required to hold page tables can be large
  - need one PTE per page in virtual address space
  - 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
  - 4 bytes/PTE = 4MB per page table
    - OS's typically have separate page tables per process
    - 25 processes = 100MB of page tables
  - solution: page the page tables (!!)

Hardware and Kernel structures for paging

- Hardware:
  - Page table base register
  - TLB (will discuss soon)
- Software:
  - Page table
    - Virtual --> physical or virtual --> disk mapping
  - Page frame database
    - One entry per physical page
    - Information on page, owning process
  - Swap file / Section list (will discuss under page replacement)

Page Frame Database



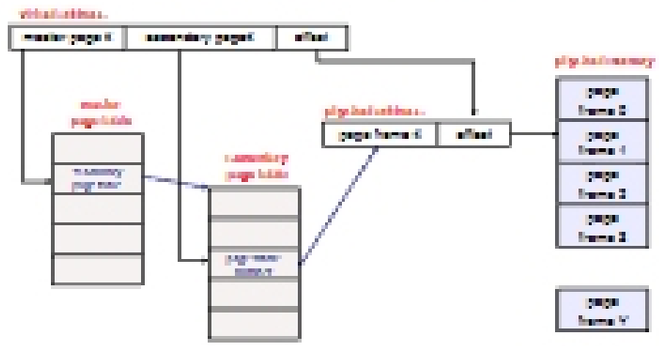
Managing Page Tables

- Last lecture:
  - size of a page table for 32 bit AS with 4KB pages was 4MB!
    - far too much overhead
  - how can we reduce this?
    - observation: only need to map the portion of the address space that is actually being used (tiny fraction of address space)
      - only need page table entries for those portions
    - how can we do this?
      - make the page table structure dynamically extensible…
  - all problems in CS can be solved with a level of indirection
    - two-level page tables

Two-level page tables

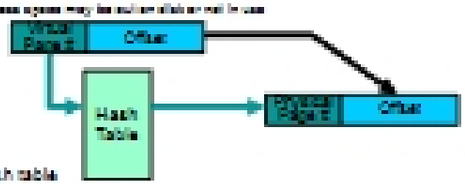
- With two-level PT's, virtual addresses have 3 parts:
  - master page number, secondary page number, offset
  - master PT maps master PN to secondary PT
  - secondary PT maps secondary PN to page frame number
  - offset + PFN = physical address
- Example:
  - 4KB pages, 4 bytes/PTE
    - how many bits in offset? need 12 bits for 4KB
  - want master PT in one page: 4KB/4 bytes = 1024 PTE
    - hence, 1024 secondary page tables
  - so: master page number = 10 bits, offset = 12 bits
    - with a 32 bit address, that leaves 10 bits for secondary PN

Two level page tables



Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware

Addressing Page Tables

- Where are page tables stored?
  - and in which address space?
- Possibility #1: physical memory
  - easy to address, no translation required
  - but, page tables consume memory for lifetime of VAS
- Possibility #2: virtual memory (OS's VAS)
  - cold (unused) page table pages can be paged out to disk
  - but, addresses page tables requires translation
    - how do we break the recursion?
  - don't page the outer page table (called wiring)
- Question: can the kernel be paged?

Generic PTE

- PTE maps virtual page to physical page
- Includes some page properties
  - Valid?, writable?, dirty?, cacheable?

Some acronyms used in this lecture:
- 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

Real Page Tables

- Design requirements
  - Minimize memory use (PT are pure overhead)
  - Fast (logically accessed on every memory ref)
- Requirements lead to
  - Compact data structures
  - O(1) access (e.g. indexed lookup, hashtable)
- Examples: X86