CS 537
Lecture 12
Paging

Michael Swift

Notes

- Writing assignment 1 has been posted - due next Thursday
- I want to meet with groups this week. I'll have a sign-up sheet after class
- Today: Paging and TLBs
- Questions from last time:
  - What is virtual memory?
  - What does it do?
  - What is it good for?

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

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page.
 */
struct page {
  unsigned long flags;               // Atomic flags: locked,referenced,dirty,slab,disk
  atomic_t _count;                   // Usage count, see below. */
  atomic_t _mapcount;                // Count of ptes mapped in mms,
                                     // to show when page is mapped
                                     // & limit reverse map searches.
struct {
      unsigned long private;          // Used for managing pages used in file I/O
      struct address_space *mapping;  // Used for memory mapped files
   };
   pgoff_t index;                     // Our offset within mapping. */
   struct list_head lru;              // Lock on Pageout list, active_list
   void *virtual;                     // Kernel virtual address *
};
```

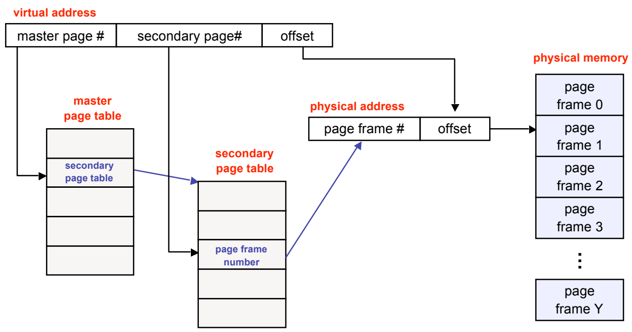

Managing Page Tables

- 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

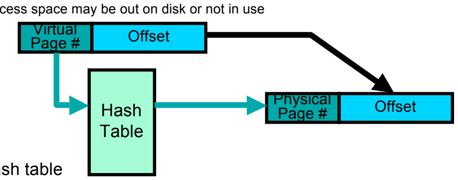
- With two-level PT’s, virtual addresses have 3 parts:
  - master page number, secondary page number, offset
  - master PT maps master PN to secondary PT
  - secondary PT maps secondary PN to page frame number
  - offset + PFN = physical address
- Example:
  - 4KB pages, 4 bytes/PTE
    - how many bits 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?

Virtual Page #	Physical Page #	Property bits

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

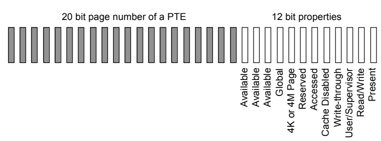
- Design requirements
  - Minimize memory use (PT are pure overhead)
  - Fast (logically accessed on every memory ref)
- Requirements lead to
  - Compact data structures
  - O(1) access (e.g. indexed lookup, hashtable)
- Examples: X86

X86-32 Address Translation

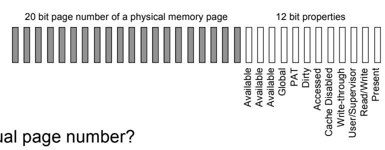
- Page tables organized as a two-level tree
  - Efficient because address space is sparse
  - Each level of the tree indexed using a piece of the virtual page number for fast lookups
- One set of page tables per process
  - Current set of page tables pointed to by CR3
- CPU walks the page tables to find translations
  - Accessed and dirty bits updated by CPU
- 4K or 4M (sometimes 2M) pages
  - Why?

X86-32 PDE and PTE Details

Page Directory Entry (PDE)



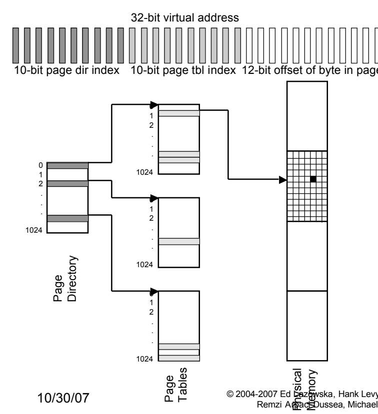
Page Table Entry (PDE)



Where is the virtual page number?
If a page is not present, all but bit 0 are available for OS

IA-32 Intel Architecture Software Developer's Manual, Volume 3, pg. 3-24

X86-32 Page Table Lookup



- Top 10 address bits index page directory and return a page directory entry that points to a page table
- Middle 10 bits index the page table that points to a physical memory page
- Bottom 12 bits are an offset to a single byte in the physical page
- Checks made at each step to ensure desired page is available in memory and that the process making the request has sufficient rights to access the page

How well does x86 work?

- How big is the minimum size page table?
- Does it support sparse address spaces well?
- Does it support paging the page table?
- How many memory lookups are required to find an entry?

Making it all efficient

- Original page table schemed doubled the cost of memory lookups
  - one lookup into page table, a second to fetch the data
- Two-level page tables triple the cost!!
  - two lookups into page table, a third to fetch the data
- How can we make this more efficient?
  - goal: make fetching from a virtual address about as efficient as fetching from a physical address
  - solution: use a hardware cache inside the CPU
    - cache the virtual-to-physical translations in the hardware
    - called a translation lookaside buffer (TLB)
    - TLB is managed by the memory management unit (MMU)

TLBs

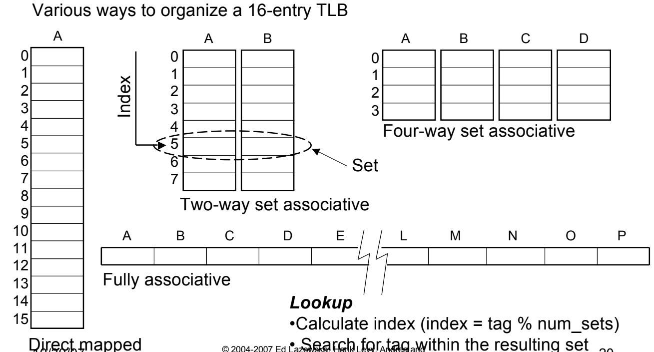
- Translation lookaside buffers
  - translates virtual page #s into PTEs (not physical addrs)
  - can be done in single machine cycle
- TLB is implemented in hardware
  - is associative cache (many entries searched in parallel)
  - cache tags are virtual page numbers
  - cache values are PTEs
  - with PTE + offset, MMU can directly calculate the PA
- TLBs exploit locality
  - processes only use a handful of pages at a time
    - 16-48 entries in TLB is typical (64-192KB for 4kb pages)
    - can hold the “hot set” or “working set” of process
  - hit rates in the TLB are therefore really important

TLB Organization

TLB Entry

Tag (virtual page number)	Value (page table entry)

Various ways to organize a 16-entry TLB



Lookup
- Calculate index (index = tag % num_sets)
- Search for tag within the resulting set
- Why not use upper bits of tag value for index?

Associativity Trade-offs

- Higher associativity
  - Better utilization, fewer collisions
  - Slower
  - More hardware
- Lower associativity
  - Fast
  - Simple, less hardware
  - Greater chance of collisions
- How does associativity affect OS behavior?
- How does page size affect TLB performance?

Managing TLBs

- Address translations are mostly handled by the TLB
  - >99% of translations, but there are TLB misses occasionally
  - in case of a miss, who places translations into the TLB?
- Hardware (memory management unit, MMU)
  - knows where page tables are in memory
    - OS maintains them, HW access them directly
  - tables have to be in HW-defined format
  - this is how x86 works
- Software loaded TLB (OS)
  - TLB miss faults to OS, OS finds right PTE and loads TLB
  - must be fast (but, 20-200 cycles typically)
    - CPU ISA has instructions for TLB manipulation
    - OS gets to pick the page table format
    - SPARC works like this

Managing TLBs (2)

- OS must ensure TLB and page tables are consistent
  - when OS changes protection bits in a PTE, it needs to invalidate the PTE if it is in the TLB (on several CPUs!)
- What happens on a process context switch?
  - remember, each process typically has its own page tables
  - need to invalidate all the entries in TLB! (flush TLB)
    - this is a big part of why process context switches are costly
  - can you think of a hardware fix to this?
- When the TLB misses, and a new PTE is loaded, a cached PTE must be evicted
  - choosing a victim PTE is called the "TLB replacement policy"
  - implemented in hardware, usually simple (e.g. LRU)

X86 TLB

- TLB management shared by processor and OS
- CPU:
  - Fills TLB on demand from page table (the OS is unaware of TLB misses)
  - Evicts entries when a new entry must be added and no free slots exist
- Operating system:
  - Ensures TLB/page table consistency by flushing entries as needed when the page tables are updated or switched (e.g. during a context switch)
  - TLB entries can be removed by the OS one at a time using the INVLPG instruction or the entire TLB can be flushed at once by writing a new entry into CR3

Example: Pentium-M TLBs

- Four different TLBs
  - Instruction TLB for 4K pages
    - 128 entries, 4-way set associative
  - Instruction TLB for large pages
    - 2 entries, fully associative
  - Data TLB for 4K pages
    - 128 entries, 4-way set associative
  - Data TLB for large pages
    - 8 entries, 4-way set associative
- All TLBs use LRU replacement policy
- Why different TLBs for instruction, data, and page sizes?

SPARC TLB

- SPARC is RISC (simpler is better) CPU
- Example of a "software-managed" TLB
  - TLB miss causes a fault, handled by OS
  - OS explicitly adds entries to TLB
  - OS is free to organize its page tables in any way it wants because the CPU does not use them
  - E.g. Linux uses a tree like X86, Solaris uses a hash table

Minimizing Flushes

- On SPARC, TLB misses trap to OS (SLOW)
  - We want to avoid TLB misses
  - Retain TLB contents across context switch
- SPARC TLB entries enhanced with a context id
  - Context id allows entries with the same VPN to coexist in the TLB (e.g. entries from different process address spaces)
  - When a process is switched back onto a processor, chances are that some of its TLB state has been retained from the last time it ran
- Some TLB entries shared (OS kernel memory)
  - Mark as global
  - Context id ignored during matching

Example:UltraSPARC III TLBs

- Five different TLBs
- Instruction TLBs
  - 16 entries, fully associative (supports all page sizes)
  - 128 entries, 2-way set associative (8K pages only)
- Data TLBs
  - 16 entries, fully associative (supports all page sizes)
  - 2 x 512 entries, 2-way set associative (each supports one page size per process)
- Valid page sizes – 8K (default), 64K, 512K, and 4M
- 13-bit context id – 8192 different concurrent address spaces
  - What happens if you have > 8192 processes?

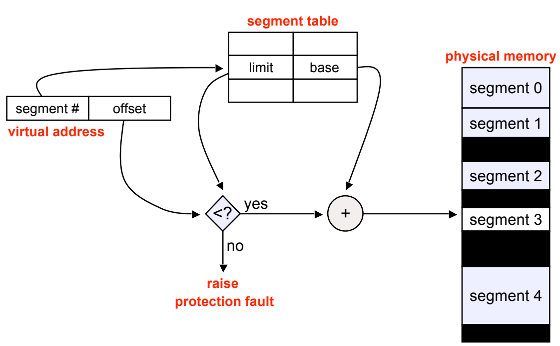
Hardware vs. Software TLBs

- Hardware benefits:
  - TLB miss handled more quickly (without flushing pipeline)
- Software benefits:
  - Flexibility in page table format
  - Easier support for sparse address spaces
  - Faster lookups if multi-level lookups can be avoided
- Intel Itanium has both!
  - Plus reverse page tables

Segmentation

- A similar technique to paging is segmentation
  - segmentation partitions memory into logical units
    - stack, code, heap, …
  - on a segmented machine, a VA is <segment #, offset>
  - segments are units of memory, from the user's perspective
- A natural extension of variable-sized partitions
  - variable-sized partition = 1 segment/process
  - segmentation = many segments/process
- Hardware support:
  - multiple base/limit pairs, one per segment
    - stored in a segment table
  - segments named by segment #, used as index into table

Segment lookups



Combining Segmentation and Paging

- Can combine these techniques
  - x86 architecture supports both segments and paging
- Use segments to manage logically related units
  - stack, file, module, heap, …?
  - segment vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed chunks
  - makes segments easier to manage within PM
    - no external fragmentation
    - segments are "pageable"- don't need entire segment in memory at same time
- Linux:
  - 1 kernel code segment, 1 kernel data segment
  - 1 user code segment, 1 user data segment
  - 1 task state segments (stores registers on context switch)
  - 1 "local descriptor table" segment (not really used)
  - all of these segments are paged

Cool Paging Tricks

- Exploit level of indirection between VA and PA
  - shared memory
    - regions of two separate processes' address spaces map to the same physical frames
      - read/write: access to share data
      - execute: shared libraries!
    - will have separate PTEs per process, so can give different processes different access privileges
    - must the shared region map to the same VA in each process?
  - copy-on-write (COW), e.g. on fork( )
    - instead of copying all pages, created shared mappings of parent pages in child address space
      - make shared mappings read-only in child space
      - when child does a write, a protection fault occurs, OS takes over and can then copy the page and resume client

Why should you care?

- Paging impacts performance
  - Managing virtual memory costs ~ 3%
- TLB management impacts performance
  - If you address more than fits in your TLB
  - If you context switch
- Page table layout impacts performance
  - Some architectures have natural amounts of data to share:
    - 4mb on x86