

CS 537 Lecture 9 Paging

Michael Swift

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

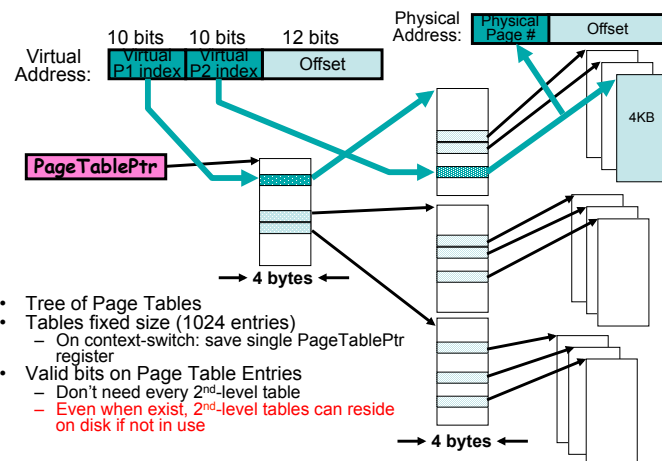
1

Multi-level Translation

- Problem: what if you have a sparse address space
 - e.g. out of 4GB, you use 1 MB spread out
 - 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**
- What about a tree of tables?
 - Upper levels: NULL pointers for unused lower levels
 - Lowest level page: translate a range of virtual addresses, or NULL for unmapped pages
- Could have any number of levels
 - x86 has 2
 - x64 has 4

2

Example two-level page table



Multi-level Translation Analysis

- Pros:
 - Only need to allocate as many page table entries as we need for application
 - In other words, sparse address spaces are easy
 - Easy memory allocation
 - Easy Sharing
 - Share at segment or page level (need additional reference counting)
- Cons:
 - One pointer per page (typically 4K – 16K pages today)
 - Two (or more, if >2 levels) lookups per reference
 - Seems very expensive!

4

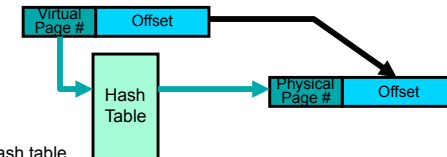
How many levels do you need?

- Each level of a multi-level page table resides on one page
 - Example: 4 KB pages, 3 bit addresses
 - 4096 bytes / 4 bytes per PTE = 1024 PTEs/page = 2^{10} pages of 2^2 KB, or 2^{12} bytes, 4096 KB 4MB mapped by one page of PTEs
 - Example: 8 KB pages, 64 bit addresses
 - 8192 bytes / 8 bytes per PTE = 1024 PTEs/page = 2^{10} pages of 2^{13} bytes (8KB) = 2^{23} bytes = 8192 KB = 8 MB
- For 32 bit addresses with 4 kb pages:
 - offset is 12 bits
 - Each page maps 2^{10} entries, or 10 more bits of address
- For 64 bit addresses with 4 kb pages
 - Offset is 12 bits
 - Each page maps 2^9 entries, need 6 levels for the remaining 52 bits

5

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!

6

Choosing a page size

- Small pages (VAX had 512 byte pages):
 - Little internal fragmentation
 - Lots of space in page tables
 - 1 gb (2^{30} bytes) takes ($2^{30}/2^9$) PTEs at 4 (2^2) bytes each = 8 MB
 - Lots of space spent caching translations
 - Fast to transfer to/from disk
- Large pages, e.g. 64 KB pages
 - Smaller page tables
 - 1 GB (2^{30} bytes) takes ($2^{30}/2^{16}$) at 4 bytes = 64 KB of page tables
 - Less space in cache for translations
 - More internal fragmentation as only part of a page is used
 - Slow to copy to/from disk

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpac-Dusseau, Michael Swift

7

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)

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpac-Dusseau, Michael Swift

8

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 */
};
```

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpacı-Dusseau, Michael Swift

9

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?

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpacı-Dusseau, Michael Swift

10

Making it all efficient

- Original page table scheme 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)

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpacı-Dusseau, Michael Swift

11

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
 - 32-128 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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpacı-Dusseau, Michael Swift

12

Example

TLB	
v:0x0100, p:0x2310,R	
V:0x0101, p:0x3210,RW	
V:0x0202, p:0x4872,RW	

cr3: 0x1010	0x00	0x8764
	0x01	0x8780
	0x02	0x9542
	0x03	invalid

0x00	0x9248
0x01	0x4432
0x02	0x2354
0x03	0x1185

0x00	0x7983
0x01	0x6752
0x02	0x6652
0x03	invalid

translate: 0x0100 432
Translate: 0x0103 743

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

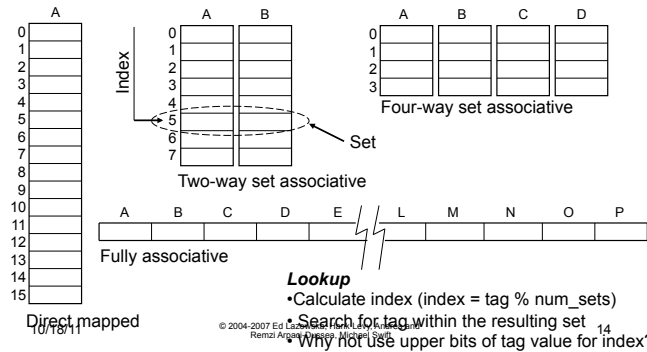
13

TLB Organization

TLB Entry

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

Various ways to organize a 16-entry TLB



Associativity Trade-offs

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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

15

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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpaci-Dusseau, Michael Swift

16

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?
- What happens when a mapping changes?
 - **Shootdown** – evict old TLB entry if it could be in use
- 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)

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dusseau, Michael Swift

17

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 or swapping out)
 - 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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dusseau, Michael Swift

18

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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dusseau, Michael Swift

19

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** (also called ASID)
 - 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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dusseau, Michael Swift

20

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

10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpac-Dusseau, Michael Swift

21

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

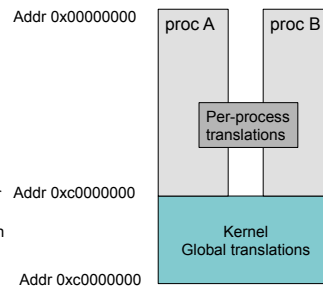
10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpac-Dusseau, Michael Swift

22

User vs Kernel addresses

- Low region of address space is private, per-process memory
- High region reserved for kernel use and has same translations for all processes
 - **Privileged** bit in PTE or TLB marks high region as only accessible when in privileged mode



10/18/11

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and
Remzi Arpac-Dusseau, Michael Swift

23