Virtual Memory

Samira Khan

Apr 27, 2017

1

Virtual Memory

- Idea: Give the programmer the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory
- Programmer can assume he/she has "infinite" amount of physical memory
- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  - Illusion is maintained for each independent process

2

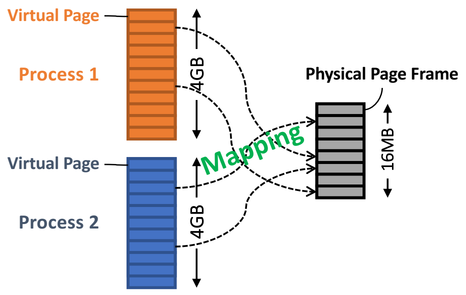
Basic Mechanism

- Indirection (in addressing)
- Address generated by each instruction in a program is a "virtual address"
  - i.e., it is not the physical address used to address main memory
- An "address translation" mechanism maps this address to a "physical address"
  - Address translation mechanism can be implemented in hardware and software together

*"At the heart [...] is the notion that 'address' is a concept **distinct** from 'physical location.'" Peter Denning*

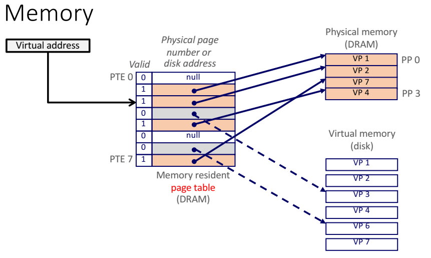
3

Overview of Paging



4

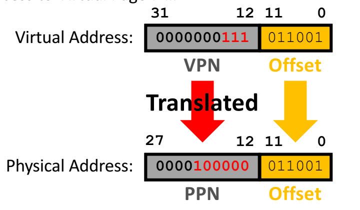
Review: Virtual Memory & Physical Memory



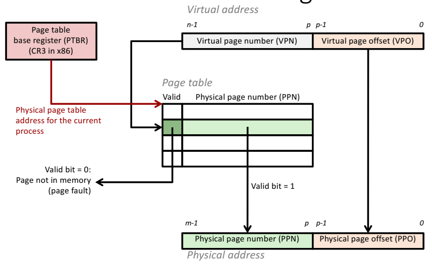
- **A *page table* contains page table entries (PTEs) that map virtual pages to physical pages.**

Translation

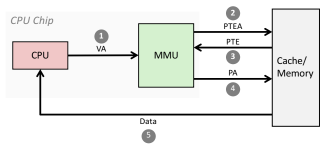
- Assume: Virtual Page 7 is mapped to Physical Page 32
- For an access to Virtual Page 7 …



Address Translation With a Page Table

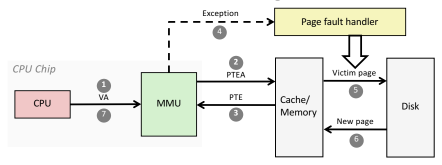

Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR) (CR3 in x86)
Physical page table address for the current process
Page table
Valid
Physical page number (PPN)
Valid bit = 0: Page not in memory (page fault)
Valid bit = 1
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

Address Translation: Page Hit



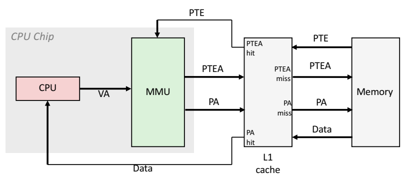
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Two Problems

- Two problems with page tables

- **Problem #1: Page table is too large**

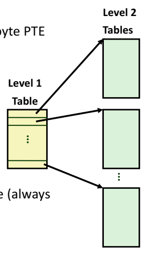
- Problem #2: Page table is stored in memory
  - Before every memory access, always fetch the PTE from the slow memory? ➔ **Large performance penalty**

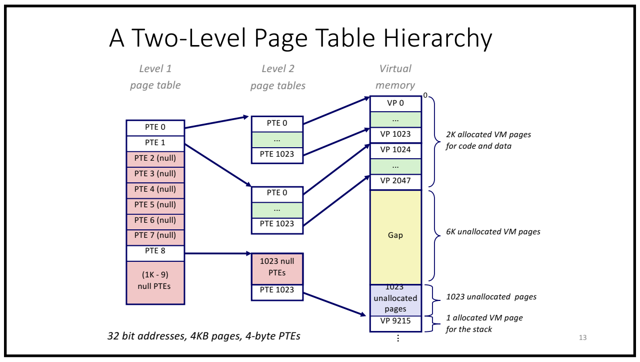
Multi-Level Page Tables

- Suppose:
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE

- Problem:
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes

- Common solution: Multi-level page table
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)

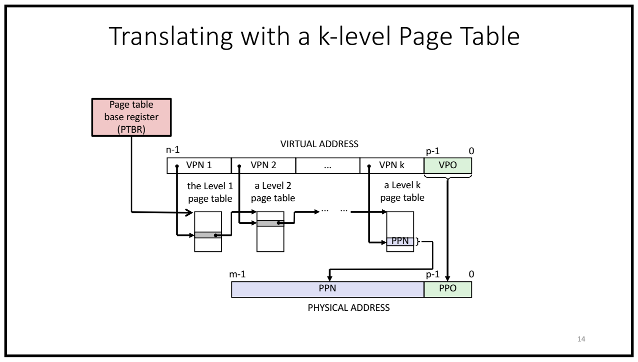




A Two-Level Page Table Hierarchy

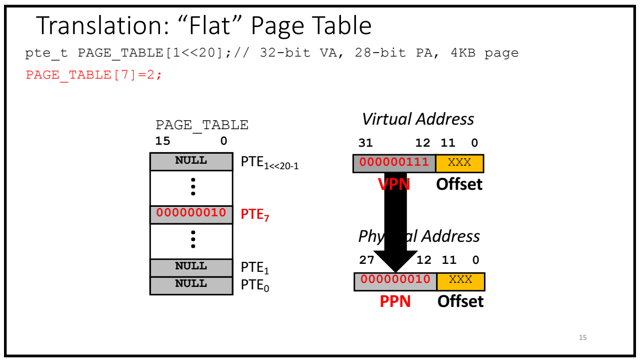
32 bit addresses, 4KB pages, 4-byte PTEs

13



Translating with a k-level Page Table

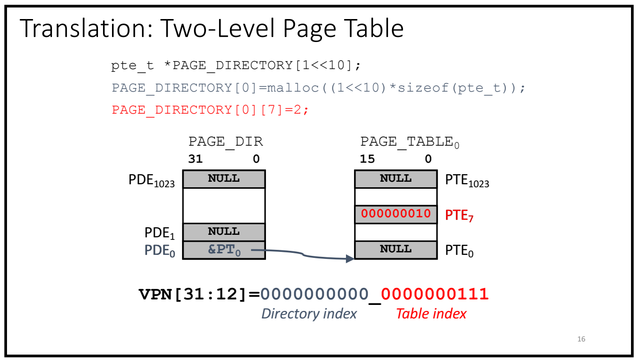
14



Translation: “Flat” Page Table

```
pte_t PAGE_TABLE[1<<20];// 32-bit VA, 28-bit PA, 4KB page
PAGE_TABLE[7]=2;
```

15



Translation: Two-Level Page Table

```
pte_t *PAGE_DIRECTORY[1<<10];
PAGE_DIRECTORY[0]=malloc((1<<10)*sizeof(pte_t));
PAGE_DIRECTORY[0][7]=2;
```

VPN[31:12]=0000000000_0000000111

Directory index *Table index*

16

Two-Level Page Table (x86)

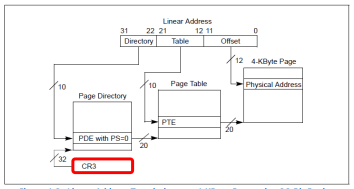


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

- **CR3**: Control Register 3 (or **Page Directory Base Register**)
  - Stores the <u>physical</u> address of the page directory
  - **Q:** Why not the virtual address?

17

Multi-Level Page Table (x86-64)

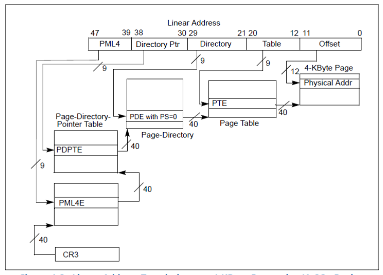
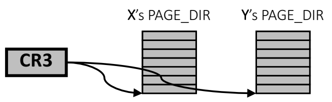


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

18

Per-Process Virtual Address Space

- Each process has its own virtual address space
  - Process **X**: text editor
  - Process **Y**: video player
  - **X** writing to its virtual address 0 <u>does not</u> affect the data stored in **Y**'s virtual address 0 (or any other address)
    - This was the entire purpose of virtual memory
  - *Each process has its own page directory and page tables*
    - On a context switch, the CR3's value must be updated



19

Two Problems

- Two problems with page tables
- **Problem #1: Page table is too large**
  - Page table has 1M entries
  - Each entry is 4B (because 4B ≈ 20-bit PPN)
  - Page table = 4MB (!!)
    - very expensive in the 80s
  - **Solution:** Hierarchical page table
- **Problem #2: Page table is in memory**
  - Before every memory access, always fetch the PTE from the slow memory? ➔ **Large performance penalty**

20

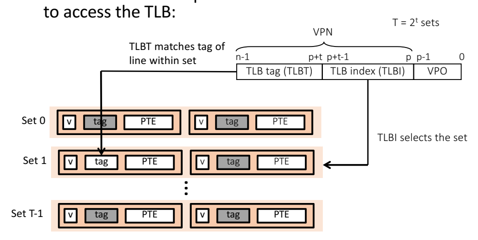
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: *Translation Lookaside Buffer* (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

21

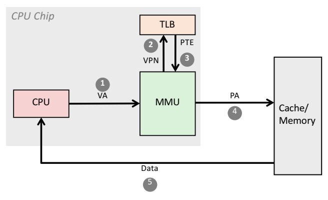
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



22

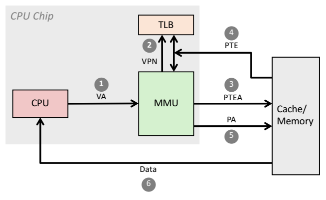
TLB Hit



A TLB hit eliminates a memory access

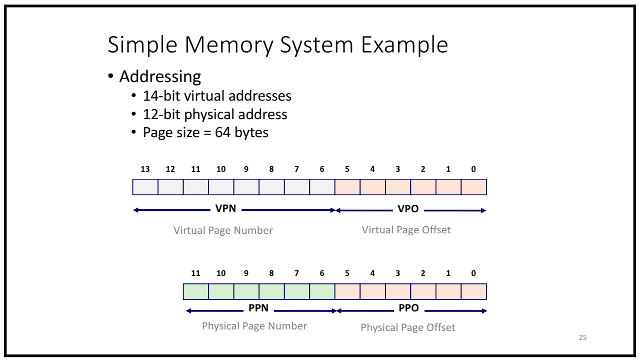
23

TLB Miss



A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?

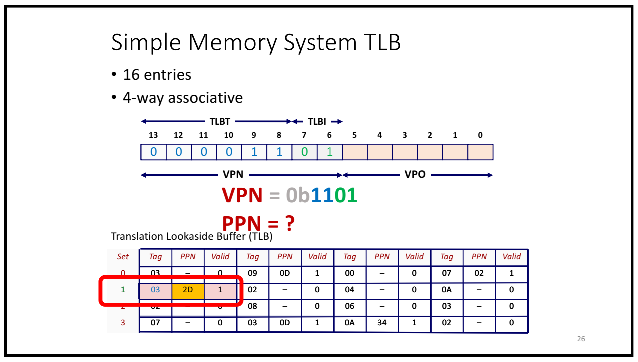
24



Simple Memory System Example

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes

25



Simple Memory System TLB

- 16 entries
- 4-way associative

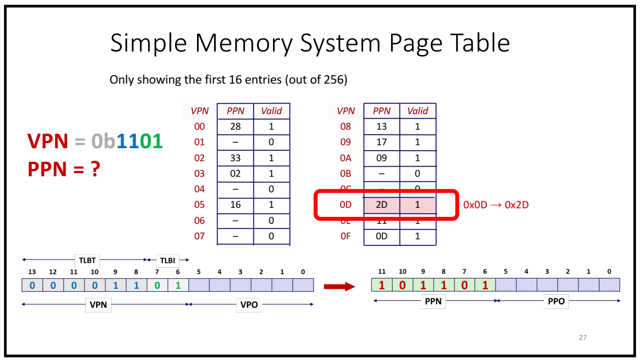
VPN = 0b1101

PPN = ?

Translation Lookaside Buffer (TLB)

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

26



Simple Memory System Page Table

Only showing the first 16 entries (out of 256)

VPN = 0b1101

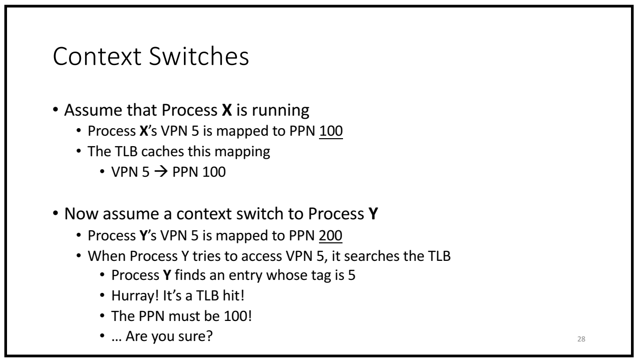
PPN = ?

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

0x0D → 0x2D

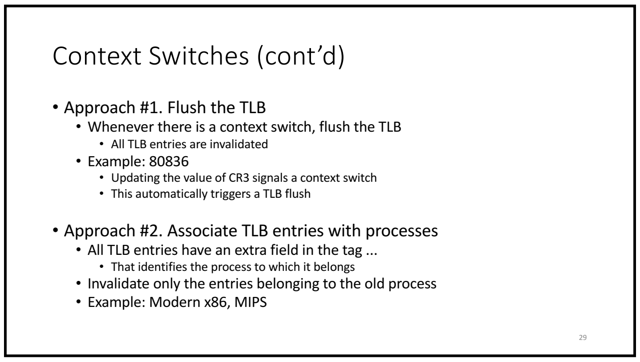
27



Context Switches

- Assume that Process **X** is running
  - Process **X**'s VPN 5 is mapped to PPN 100
  - The TLB caches this mapping
    - VPN 5 → PPN 100
- Now assume a context switch to Process **Y**
  - Process **Y**'s VPN 5 is mapped to PPN 200
  - When Process Y tries to access VPN 5, it searches the TLB
    - Process **Y** finds an entry whose tag is 5
    - Hurray! It's a TLB hit!
    - The PPN must be 100!
    - … Are you sure?

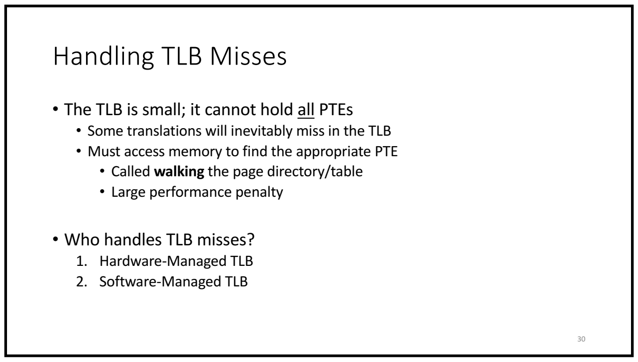
28



Context Switches (cont'd)

- Approach #1. Flush the TLB
  - Whenever there is a context switch, flush the TLB
    - All TLB entries are invalidated
  - Example: 80836
    - Updating the value of CR3 signals a context switch
    - This automatically triggers a TLB flush
- Approach #2. Associate TLB entries with processes
  - All TLB entries have an extra field in the tag ...
    - That identifies the process to which it belongs
  - Invalidate only the entries belonging to the old process
  - Example: Modern x86, MIPS

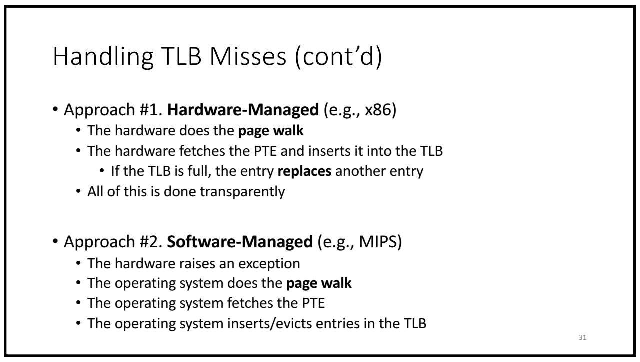
29



Handling TLB Misses

- The TLB is small; it cannot hold <u>all</u> PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called **walking** the page directory/table
    - Large performance penalty
- Who handles TLB misses?
  1. Hardware-Managed TLB
  2. Software-Managed TLB

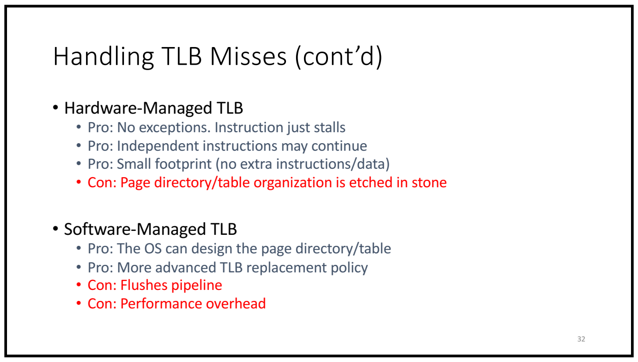
30



Handling TLB Misses (cont'd)

- Approach #1. **Hardware-Managed** (e.g., x86)
  - The hardware does the **page walk**
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - All of this is done transparently
- Approach #2. **Software-Managed** (e.g., MIPS)
  - The hardware raises an exception
  - The operating system does the **page walk**
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

31



Handling TLB Misses (cont'd)

- Hardware-Managed TLB
  - Pro: No exceptions. Instruction just stalls
  - Pro: Independent instructions may continue
  - Pro: Small footprint (no extra instructions/data)
  - Con: Page directory/table organization is etched in stone
- Software-Managed TLB
  - Pro: The OS can design the page directory/table
  - Pro: More advanced TLB replacement policy
  - Con: Flushes pipeline
  - Con: Performance overhead

32

Address Translation and Caching

- When do we do the address translation?
  - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache
- What are the issues with a virtually addressed cache?
- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

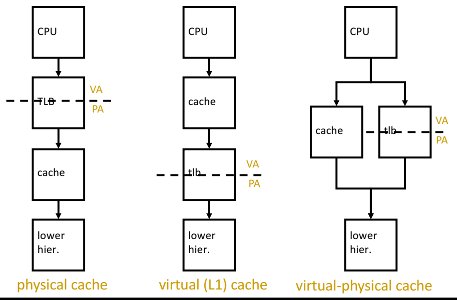
33

Homonyms and Synonyms

- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes
- Synonym: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, …
- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

34

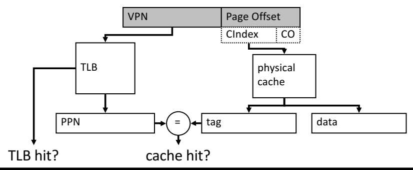
Cache-VM Interaction



35

Virtually-Indexed Physically-Tagged

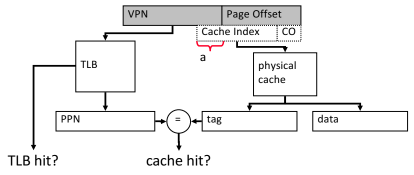
- If C≤(page_size × associativity), the cache index bits come only from page offset (***same in VA and PA***)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



36

Virtually-Indexed Physically-Tagged

- If C>(page_size × associativity), the cache index bits include VPN ⟹ Synonyms can cause problems
  - The same physical address can exist in two locations
- Solutions?



Sanity Check

- Core 2 Duo: 32 KB, 8-way set associative, page size ≥ 4K
- Cache size ≤(page_size × associativity)?
- 2^P = 4K P = 12
  - Needs 12 bits for page offset
- 2^C = 32KB, C = 15
  - Needs 15 bits to address a byte in the cache
- 2^A = 8-way, A = 3
  - *Increasing the associativity of the cache reduces the number of address bits needed to index into the cache*
  - Needs 12 bits for cache index and offset, as tags are matched for blocks in the same set
- C ≤ P + A ?
  15 ≤ 12+3? ***True***

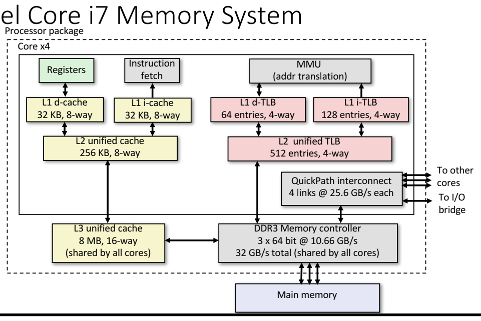
Some Solutions to the Synonym Problem

- Limit cache size to (page size times associativity)
  - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
  - make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

Today

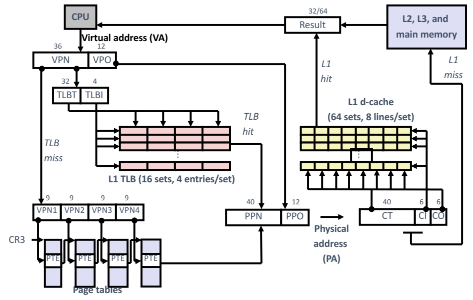
- Case study: Core i7/Linux memory system

Intel Core i7 Memory System


Processor package
Core x4
Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L1 i-cache 32 KB, 8-way
L1 d-TLB 64 entries, 4-way
L1 i-TLB 128 entries, 4-way
L2 unified cache 256 KB, 8-way
L2 unified TLB 512 entries, 4-way
QuickPath interconnect 4 links @ 25.6 GB/s each
To other cores
To I/O bridge
L3 unified cache 8 MB, 16-way (shared by all cores)
DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s 32 GB/s total (shared by all cores)
Main memory

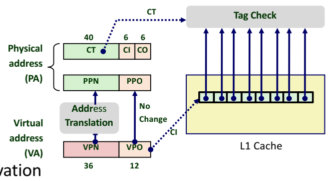
41

End-to-end Core i7 Address Translation



42

Speeding Up L1 Access



- Observation
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Generally we hit in TLB, so PPN bits (CT bits) available next
  - ***"Virtually indexed, physically tagged"***
  - Cache carefully sized to make this possible

Core i7 Level 1-3 Page Table Entries

63	62 … 52	51 … 12	11 … 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

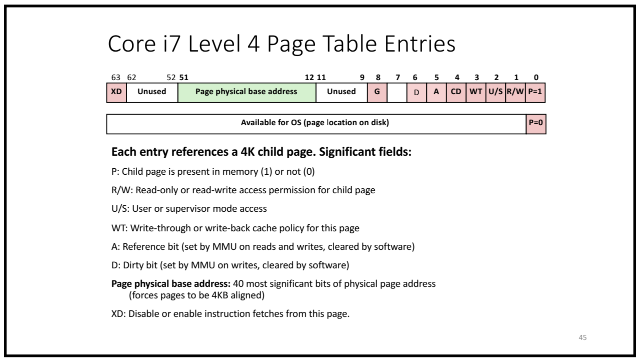
A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

44



Core i7 Level 4 Page Table Entries

63	62–52	51–12	11–9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1

Available for OS (page location on disk)	P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

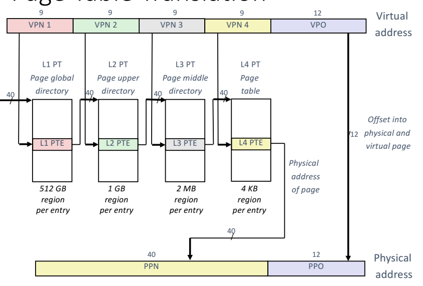
D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

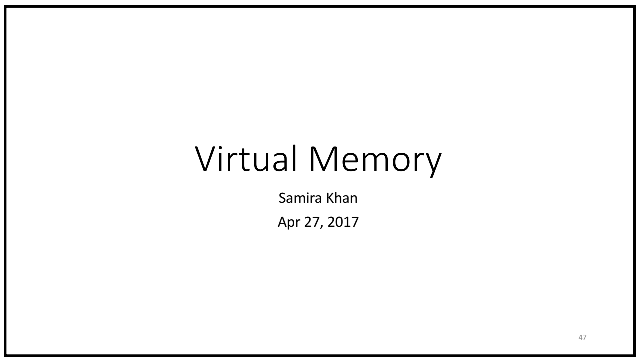
XD: Disable or enable instruction fetches from this page.

45

Core i7 Page Table Translation



46



Virtual Memory

Samira Khan

Apr 27, 2017

47