15-213

"The course that gives CMU its Zip!"

Virtual Memory
November 2, 2007

Topics

- Motivations for virtual memory
- Address translation
- Accelerating translation with TLBs

class19.ppt

Why Virtual Memory?

(1) VM uses main memory efficiently

- Main memory is a cache for the contents of a virtual address space stored on disk.
- Keep only active areas of virtual address space in memory
- Transfer data back and forth as needed.

(2) VM simplifies memory management

- Each process gets the same linear address space.

(3) VM protects address spaces

- One process can't interfere with another.
  - Because they operate in different address spaces.
- User process cannot access privileged information
  - Different sections of address spaces have different permissions.

Motivation 1: DRAM a "Cache" for Disk

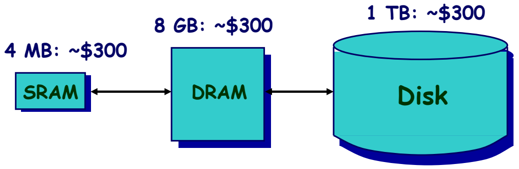
The full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

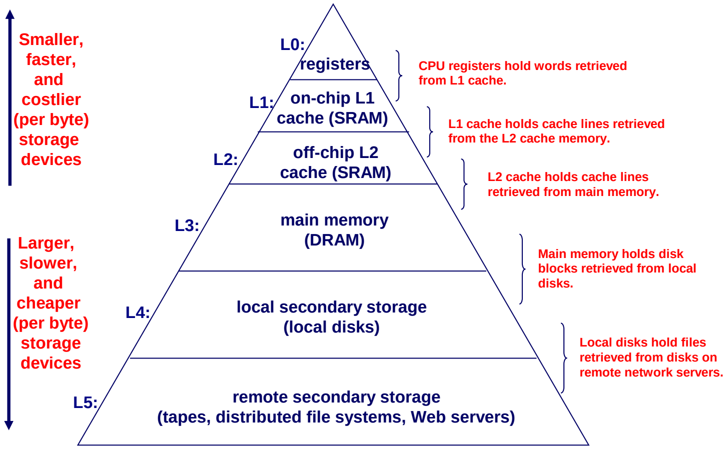
Disk storage is ~100X cheaper than DRAM storage

- 1 TB of DRAM: ~ $30,000
- 1 TB of disk: ~ $300

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Levels in Memory Hierarchy


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
registers
L1:
on-chip L1 cache (SRAM)
L2:
off-chip L2 cache (SRAM)
L3:
main memory (DRAM)
L4:
local secondary storage (local disks)
L5:
remote secondary storage (tapes, distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache.

L1 cache holds cache lines retrieved from the L2 cache memory.

L2 cache holds cache lines retrieved from main memory.

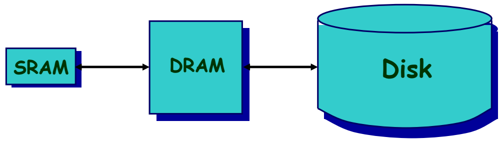
Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote network servers.

DRAM vs. SRAM as a "Cache"

DRAM vs. disk is more extreme than SRAM vs. DRAM

- access latencies:
  - DRAM is ~10X slower than SRAM
  - disk is ~100,000X slower than DRAM
- importance of exploiting spatial locality:
  - first byte is ~100,000X slower than successive bytes on disk
    - » vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- "cache" size:
  - main memory is ~1000X larger than an SRAM cache
- addressing for disk is based on sector address, not memory address



Impact of These Properties on Design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

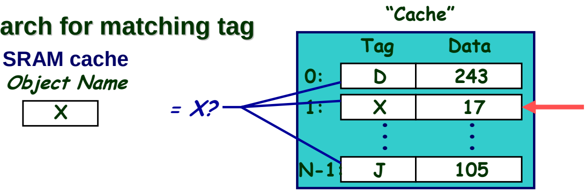
What would the impact of these choices be on:

- miss rate
- hit time
- miss latency
- tag overhead

Locating an Object in a "Cache"

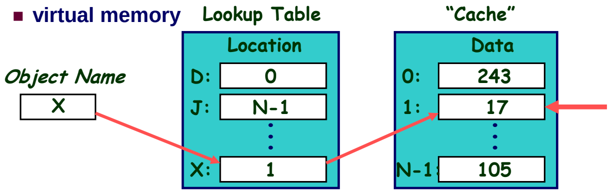
1. Search for matching tag

- SRAM cache



2. Use indirection to look up actual object location

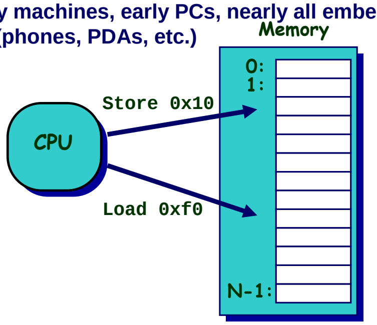
- virtual memory



A System with Physical Memory Only

Examples:

- most Cray machines, early PCs, nearly all embedded systems (phones, PDAs, etc.)

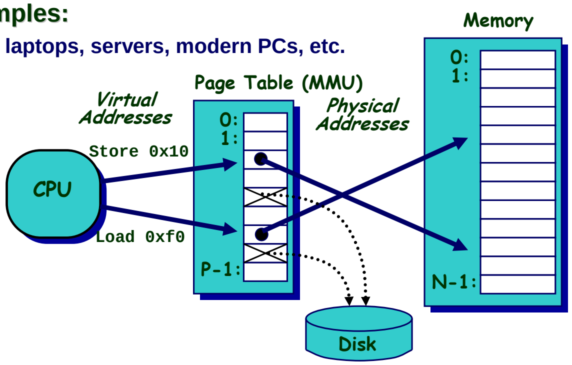


CPU's load or store addresses used directly to access memory.

A System with Virtual Memory

Examples:

- laptops, servers, modern PCs, etc.

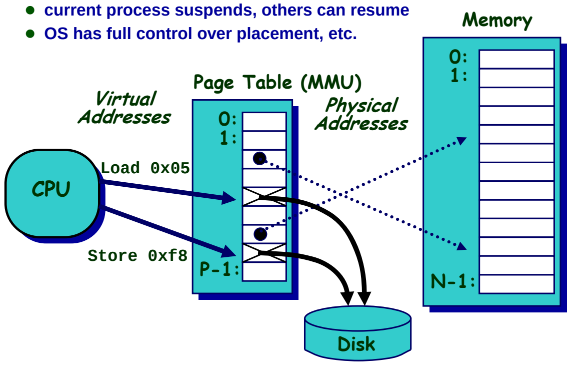


Address Translation: the hardware converts *virtual addresses* into *physical addresses* via an OS-managed lookup table (*page table*)

Page Faults (Similar to "Cache Misses")

What if an object is on disk rather than in memory?

- Page table entry indicates that the virtual address is not in memory
- An OS trap handler is invoked, moving data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.



Servicing a Page Fault

(1) Processor signals controller

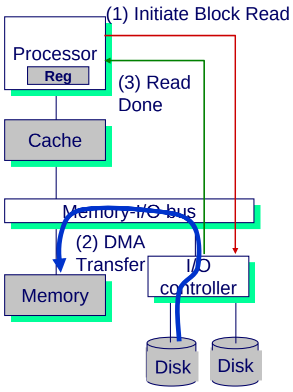
- Read block of length P starting at disk address X and store starting at memory address Y

(2) Read occurs

- Direct Memory Access (DMA)
- Under control of I/O controller

(3) Controller signals completion

- Interrupt processor
- OS resumes suspended process



Locality to the Rescue

Virtual memory works because of locality.

At any point in time, programs tend to access a set of active virtual pages called the ***working set***.

- Programs with better temporal locality will have smaller working sets.

If working set size < main memory size

- Good performance after initial compulsory misses.

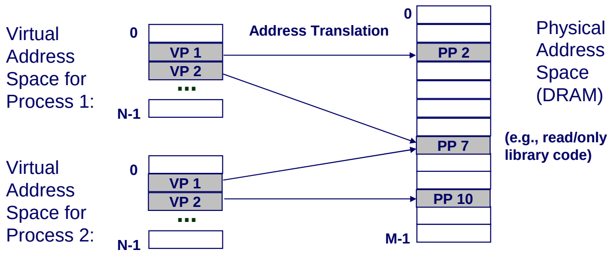
If working set size > main memory size

- ***Thrashing:*** Performance meltdown where pages are swapped (copied) in and out continuously

(2) VM as a Tool for Memory Mgmt

Key idea: Each process has its own virtual address space

- Simplifies memory allocation, sharing, linking, and loading.


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address Translation
0
PP 2
PP 7
PP 10
M-1
Physical Address Space (DRAM)
(e.g., read/only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

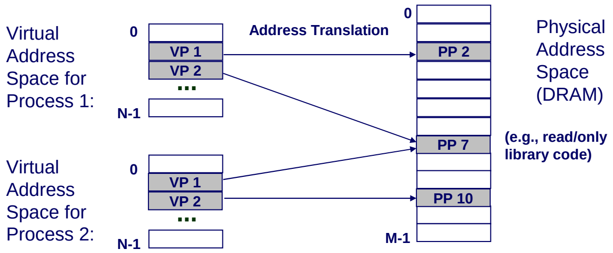
Simplifying Sharing and Allocation

Sharing code and data among processes

- Map virtual pages to the same physical page (PP 7)

Memory allocation

- Virtual page can be mapped to any physical page


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address Translation
0
PP 2
PP 7
PP 10
M-1
Physical Address Space (DRAM)
(e.g., read/only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

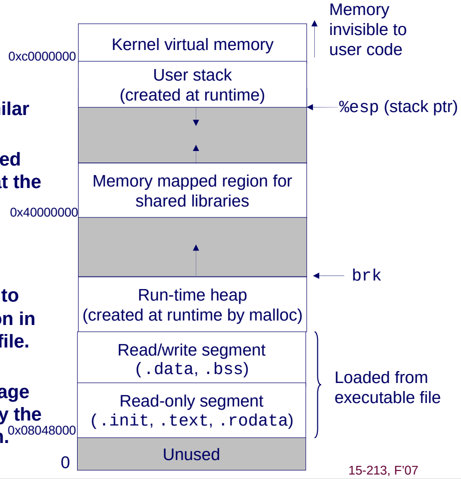
Simplifying Linking and Loading

Linking

- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address.

Loading

- `execve()` maps PTEs to the appropriate location in the executable binary file.
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system.


Memory invisible to user code
Kernel virtual memory
0xc0000000
User stack (created at runtime)
%esp (stack ptr)
Memory mapped region for shared libraries
0x40000000
brk
Run-time heap (created at runtime by malloc)
Read/write segment (.data, .bss)
Loaded from executable file
Read-only segment (.init, .text, .rodata)
0x08048000
Unused
0

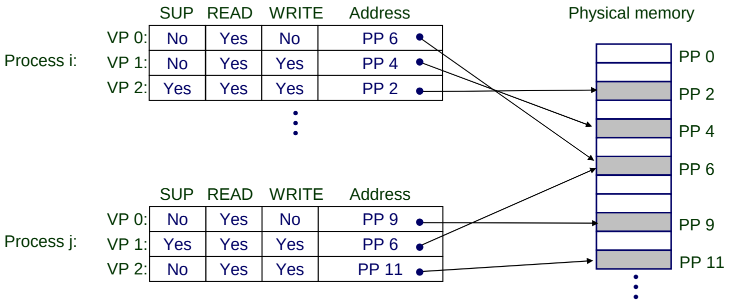
(3)VM as a Tool for Memory Protection

Extend PTEs with permission bits.

Page fault handler checks these before remapping.

- If violated, send process SIGSEGV (segmentation fault)

Page tables with permission bits



Process i:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2
⋮				

Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

Address Spaces

A *linear address space* is an ordered set of contiguous nonnegative integer addresses:

{0, 1, 2, 3, … }

A *virtual address space* is a set of $N = 2^n$ *virtual addresses*:

{0, 1, 2, …, N-1}

A *physical address space* is a set of $M = 2^m$ (for convenience) *physical addresses*:

{0, 1, 2, …, M-1}

In a system based on virtual addressing, each byte of main memory has a virtual address *and* a physical address.

VM Address Translation

Virtual Address Space

- V = {0, 1, …, N–1}

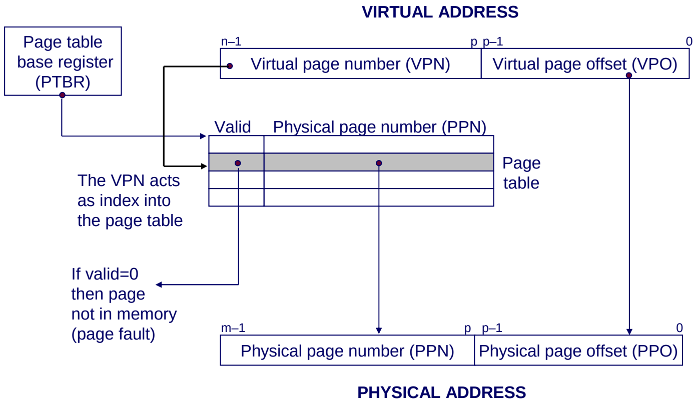
Physical Address Space

- P = {0, 1, …, M–1}
- M < N (usually, but >=4 Gbyte on an IA32 possible)

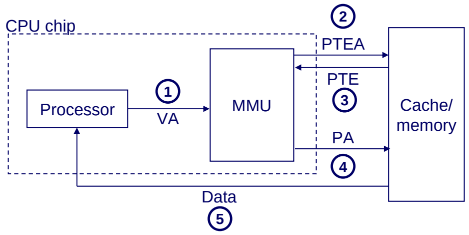
Address Translation

- MAP: $V \rightarrow P \cup \{\varnothing\}$
- For virtual address a:
  - MAP(a) = a' if data at virtual address a at physical address a' in P
  - MAP(a) = ∅ if data at virtual address a not in physical memory
    - » Either invalid or stored on disk

Address Translation with a Page Table


VIRTUAL ADDRESS
Page table base register (PTBR)
n–1
p p–1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Valid
Physical page number (PPN)
Page table
The VPN acts as index into the page table
If valid=0 then page not in memory (page fault)
m–1
p p–1
0
Physical page number (PPN)
Physical page offset (PPO)
PHYSICAL ADDRESS

Address Translation: Page Hit

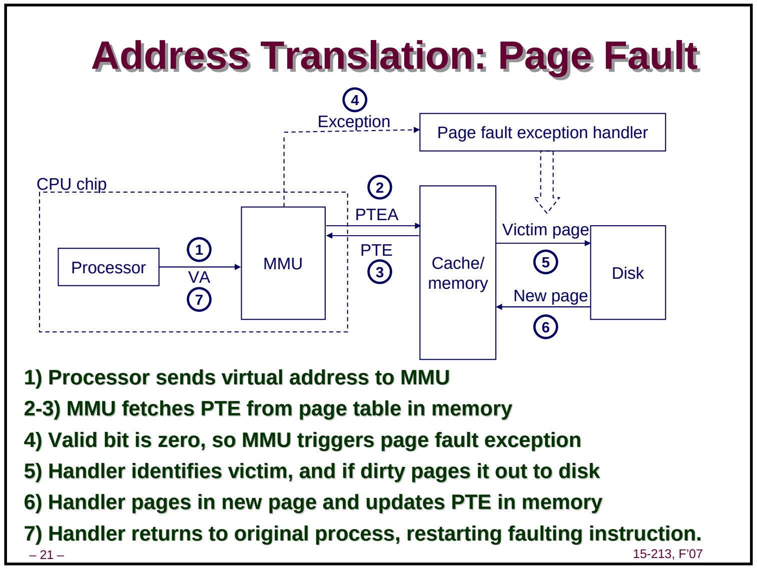


1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to L1 cache

5) L1 cache sends data word to processor



Address Translation: Page Fault

1) Processor sends virtual address to MMU

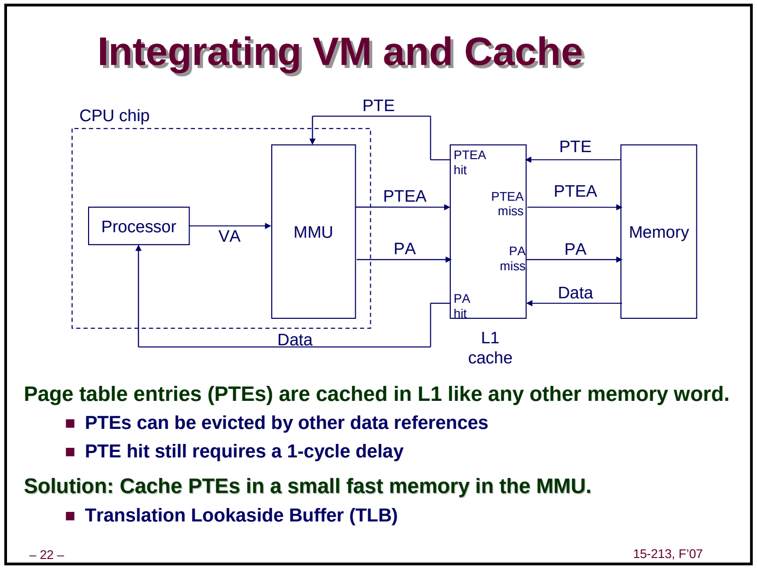
2-3) MMU fetches PTE from page table in memory

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.



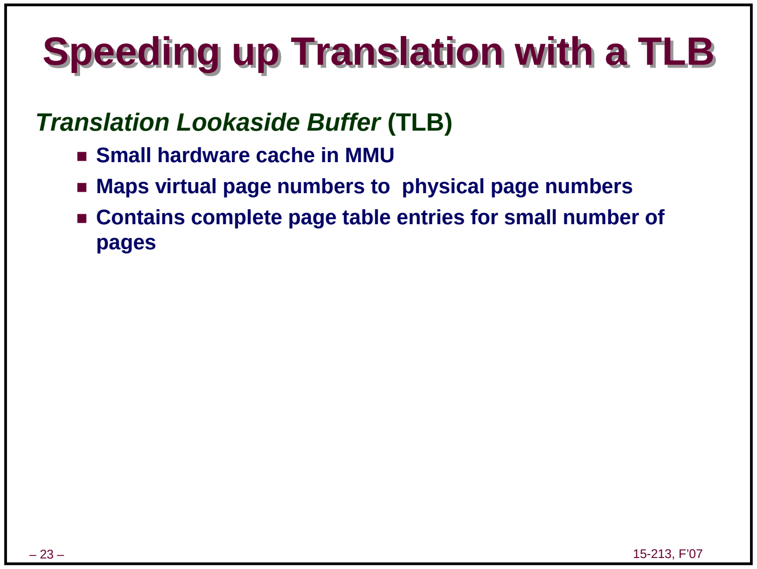
Integrating VM and Cache

Page table entries (PTEs) are cached in L1 like any other memory word.

- PTEs can be evicted by other data references
- PTE hit still requires a 1-cycle delay

Solution: Cache PTEs in a small fast memory in the MMU.

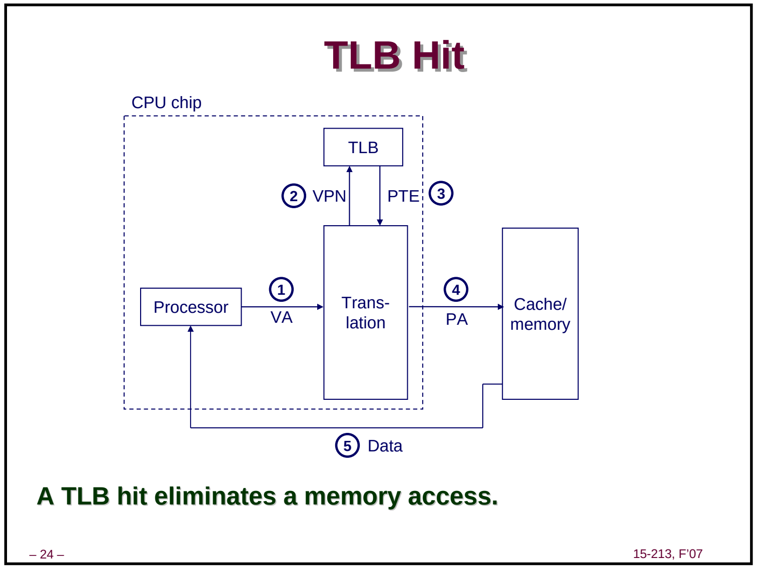
- Translation Lookaside Buffer (TLB)



Speeding up Translation with a TLB

Translation Lookaside Buffer (TLB)

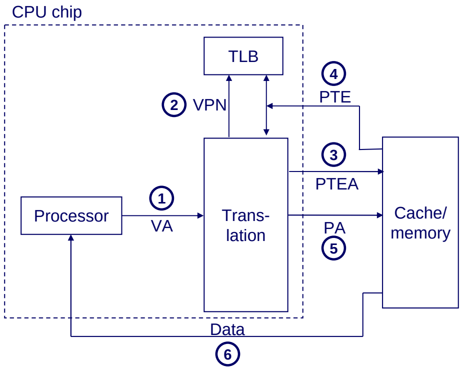
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages



TLB Hit

A TLB hit eliminates a memory access.

TLB Miss



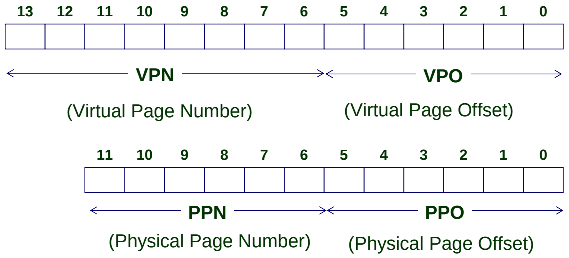
A TLB miss incurs an additional memory access (the PTE).

Fortunately, TLB misses are rare. Why?

Simple Memory System Example

Addressing

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



Simple Memory System Page Table

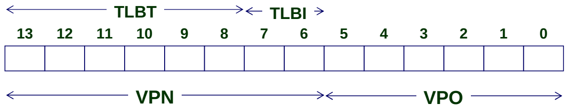
- **Only show first 16 entries (out of 256)**

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

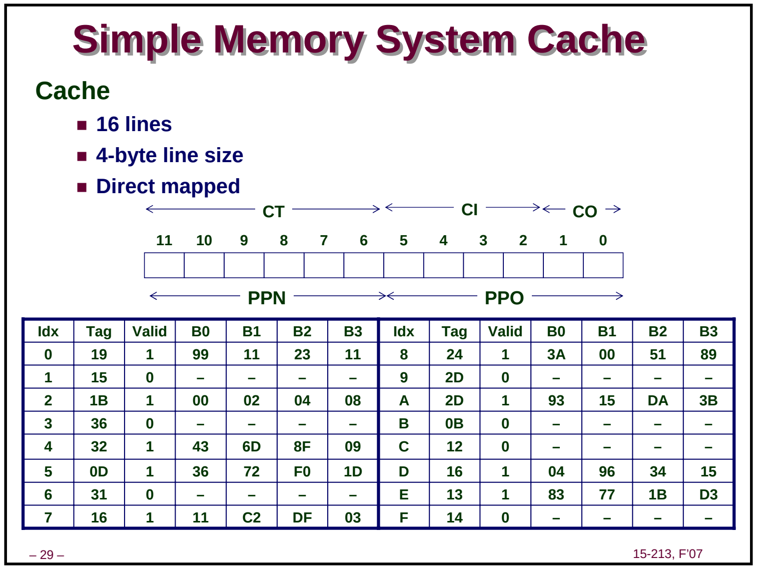
Simple Memory System TLB

TLB

- **16 entries**
- **4-way associative**



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



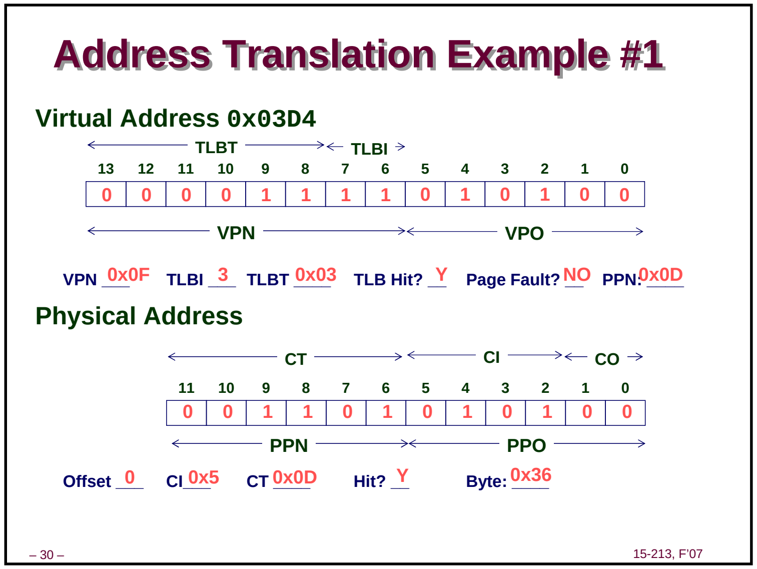
Simple Memory System Cache

Cache

- 16 lines
- 4-byte line size
- Direct mapped

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–



Address Translation Example #1

Virtual Address 0x03D4

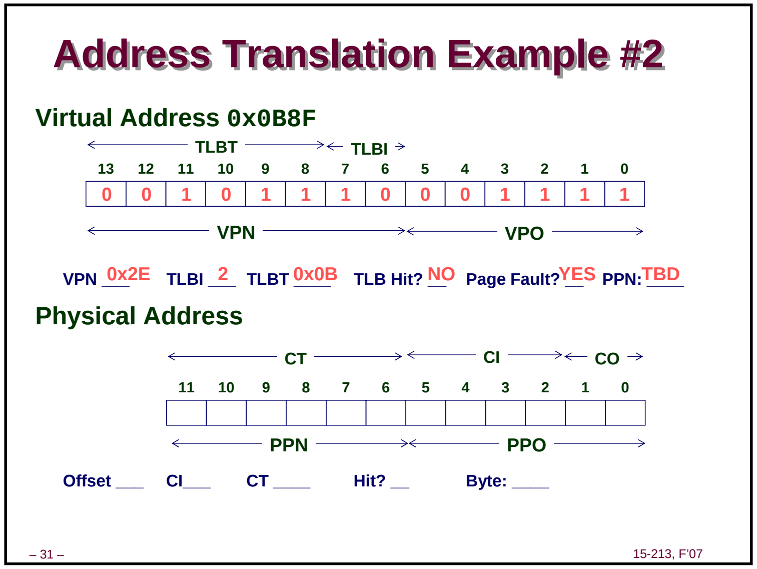
TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? NO PPN: 0x0D

Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

Offset 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36



Address Translation Example #2

Virtual Address 0x0B8F

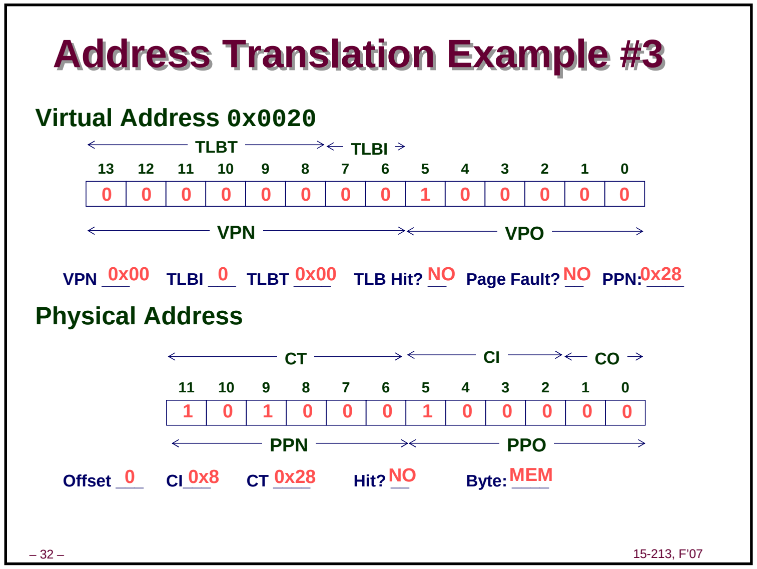
TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1
VPN								VPO					

VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? NO Page Fault? YES PPN: TBD

Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

Offset ___ CI___ CT ____ Hit? __ Byte: ____



Address Translation Example #3

Virtual Address 0x0020

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0
VPN								VPO					

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? NO Page Fault? NO PPN: 0x28

Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0
PPN						PPO					

Offset 0 CI 0x8 CT 0x28 Hit? NO Byte: MEM

Multi-Level Page Tables

Given:

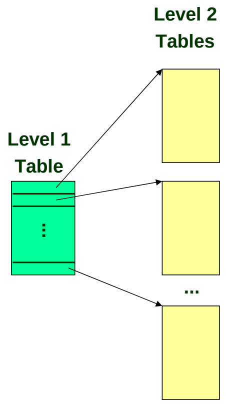
- 4KB (2^{12}) page size
- 48-bit address space
- 4-byte PTE

Problem:

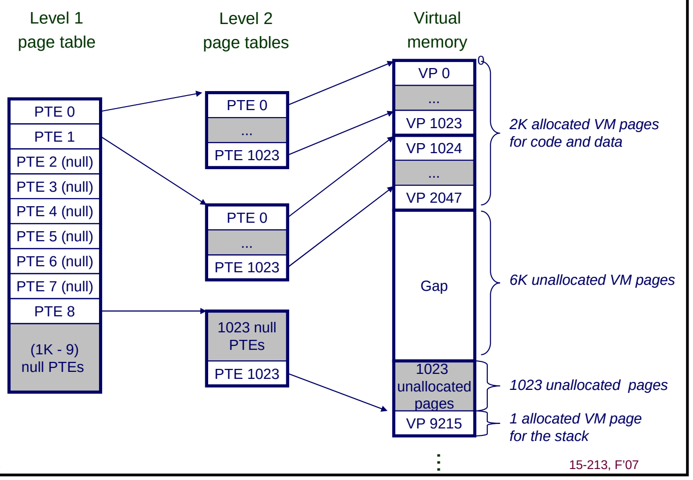
- Would need a 256 GB page table!
  - $2^{48} * 2^{-12} * 2^{2} = 2^{38}$ bytes

Common solution

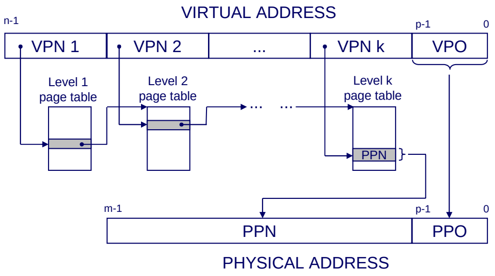
- Multi-level page tables
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (memory resident)
  - Level 2 table: Each PTE points to a page (paged in and out like other data)



A Two-Level Page Table Hierarchy



Translating with a k-level Page Table



Summary

Programmer's View of Virtual Memory

- Each process has its own private linear address space
- Cannot be corrupted by other processes

System View of Virtual Memory

- Uses memory efficiently by caching virtual memory pages stored on disk.
  - Efficient only because of locality
- Simplifies memory management in general, linking, loading, sharing, and memory allocation in particular.
- Simplifies protection by providing a convenient interpositioning point to check permissions.