CS 33 Week 9

Section 1G, Spring 2015
Prof. Eggert (TA: Eric Kim)
v1.0

Announcements

- Lab 4 due next week
  - June 3rd (Wednesday)
- Final Exam in 2 weeks!
  - Final Examination Code: 15 - Thursday, June 11, 2015, 8:00am-11:00am
  - **Location**: Ackerman Grand Ballroom
  - Cumulative (emphasis on post-MT2)

Overview

- Virtual Memory
- Linking

Virtual Memory

Tip: Read the textbook! (Chapter 9)

Pretty detail-oriented section.

Google is your friend.

Virtual Memory: Motivation

Multiple user programs need to use RAM.

Problem: Only one DRAM unit on machine.

Solution: Use Virtual Memory to let programs think they have access to entire DRAM.

Virtual Memory: High Level

A program's memory is an array with 2^N elements that resides on disk.

Partition memory into equal-sized chunks: "pages"

Use DRAM as cache for program's memory (ie a cache for disk).

Virtual Address Spaces

Each user process has its own virtual address space. The virtual addresses from P1 do not interfere with the vaddrs of P2.

Ex: Suppose P1,P2 both write to vaddr 0x7fffabcd. In general, P1 and P2 will write to different *physical* locations.

Virtual Address Spaces

Note: There are ways for different processes to *share* pages, ie for two processes to write to the same physical memory location. (mmap)

Physical vs Virtual

Physical Address Space: Refers to addressable bytes on the DRAM installed in the machine.

Virtual Address Space: Refers to addressable bytes that may reside in DRAM or on disk. The address space that each program owns.

Page Tables

Maps vaddr to actual location, ie either DRAM or disk.

Page Table Entry (PTE): Rows of page table.

In practice, PTE's include more information (such as NX-bit, permissions, etc.)

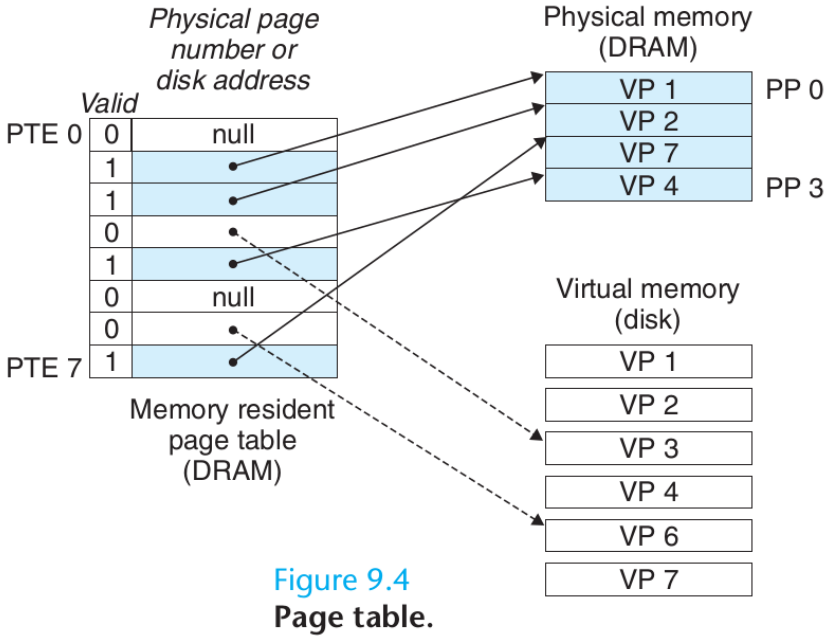


Figure 9.4
Page table.

Page Tables

Resides in DRAM.

The special CR3 register stores the start location of the page table. (register lives in MMU)

Each page stores many addresses. (Ex: 1 page stores 4 KiB = (4*2^10) addresses)

Page Tables

Can nest page tables too:

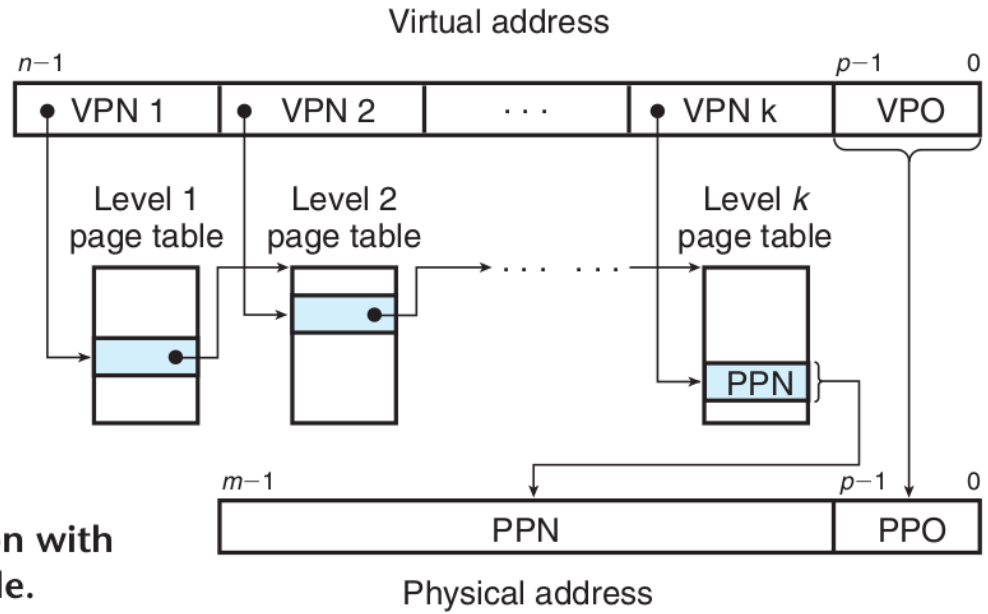


Figure 9.18
Address translation with a k-level page table.

Using Page Tables

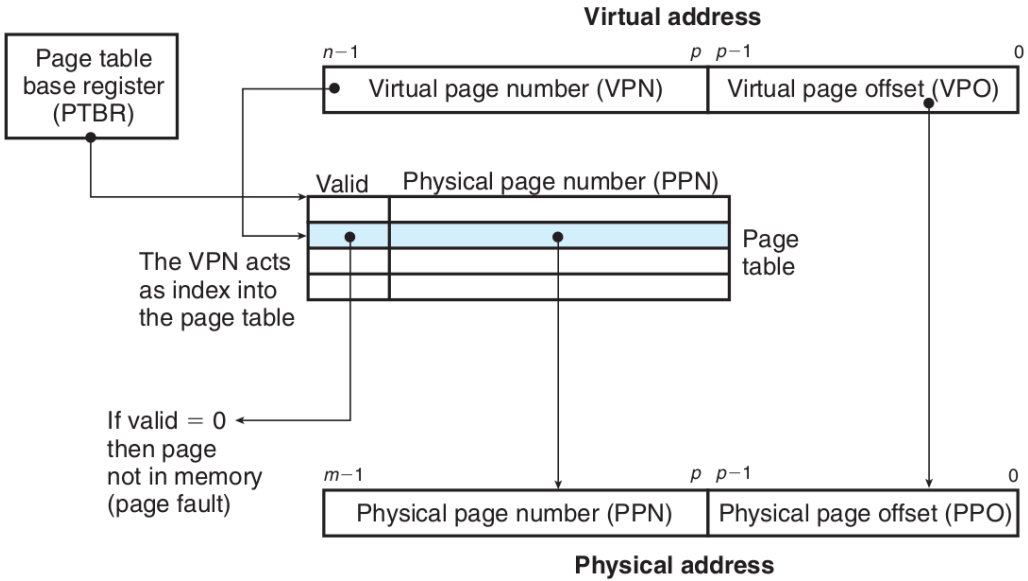


Figure 9.12 **Address translation with a page table.**

(1) Split vaddr into VPN and VPO

(2) Use VPN as index into page table.

If not valid: **Page fault!**

Else:

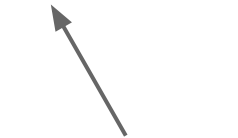
(3) Grab PPN from table, and construct paddr as:

paddr = PPN | VPO

concatenate

Page Fault

(1) Split vaddr into VPN and VPO
(2) Use VPN as index into page table.
    If not valid: **Page fault!**
    Else:
(3) Grab PPN from table, and construct paddr as:
    paddr = PPN | VPO



If the PTE for a vaddr is invalid, then the physical page PPN is not loaded in DRAM.

Must ask OS to:

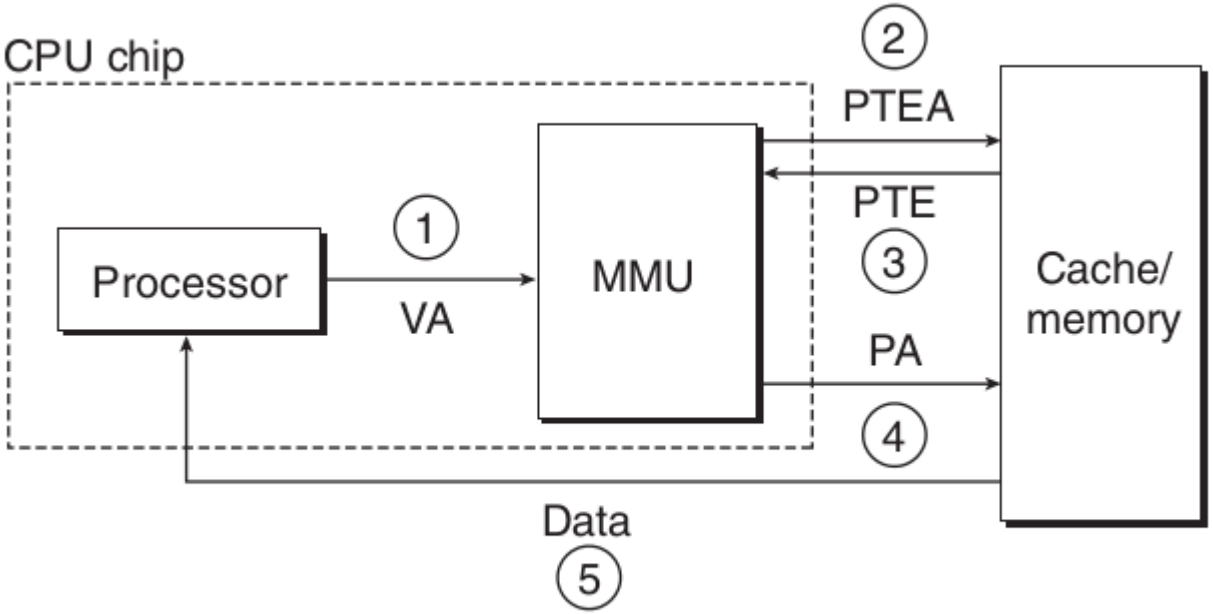
(1) Evict a victim page from DRAM

(2) Load the physical page PPN to DRAM

(3) Re-run the instruction that generated the page fault.

Figure: Page Hit

VA: vaddr
PA: paddr
PTE: Page Table Entry
PTEA: Page Table Entry Address



(a) Page hit

Figure: Page Fault

VA: vaddr
PA: paddr
PTE: Page Table Entry
PTEA: Page Table Entry Address

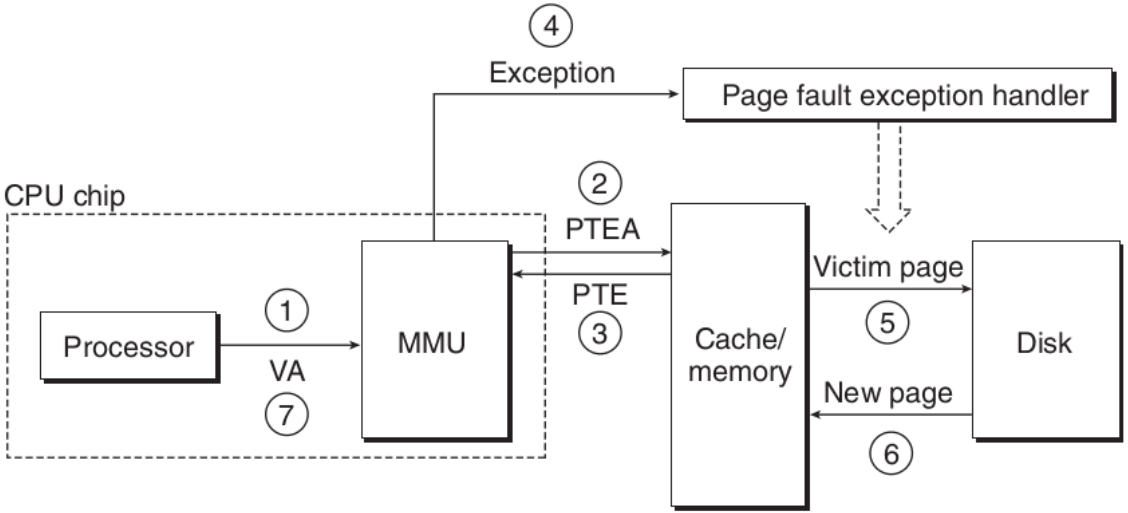


Figure 9.13 **Operational view of page hits and page faults.** VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

Paging: Disk I/O

Suppose process P1 gets a page fault. Swapping in the new page (and swapping out victim page) is fairly expensive, due to disk I/O.

So, OS will likely run other processes while doing the page swaps.

Question: Suppose P1,P2 are running processes. Can you think of a scenario where a program P1 gets "stuck"?

Answer: Thrashing! Suppose the first page fault evicted a page that P2 needed. When P2 runs, it will issue its own page fault. Suppose the second page fault evicts a page that P1 needs. When P1 resumes, it will issue another page fault.
...Rinse and repeat...

MMU: Memory Management Unit

MMU is a special chip in the CPU that performs address translation.

Contains hardware to do vaddr->paddr mapping.

Goal: MMU must be as fast as possible.

TLB: Cache for MMU

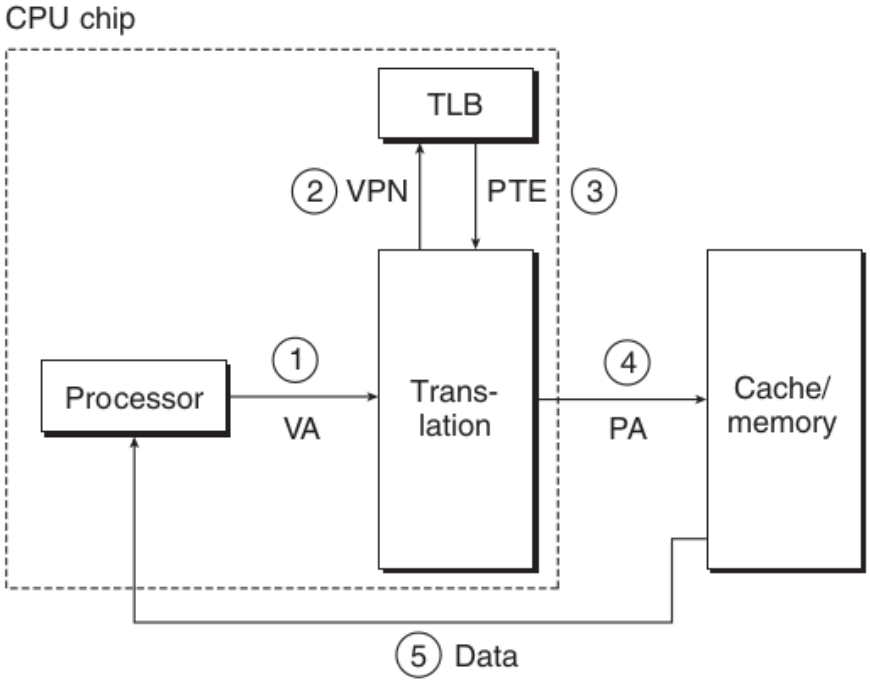
Short for: "Translation Lookaside Buffer"

Caches mappings: VPN -> PPN

Typically has high associativity (ie 4-way set associative).

TLB: Hit

VA: vaddr
PA: paddr
PTE: Page Table Entry
PTEA: Page Table Entry Address



(a) TLB hit

TLB: Miss

VA: vaddr
PA: paddr
PTE: Page Table Entry
PTEA: Page Table Entry Address

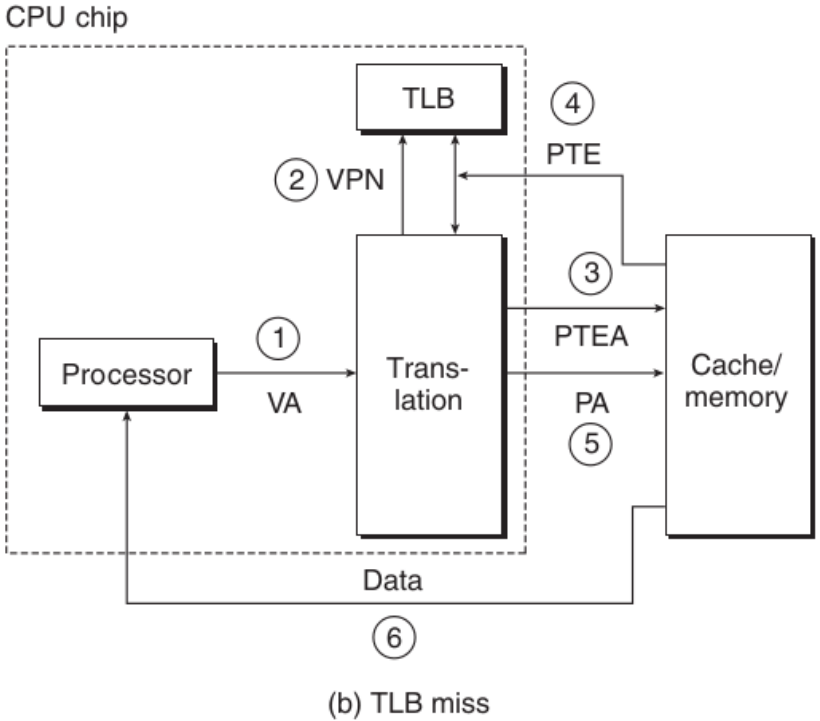
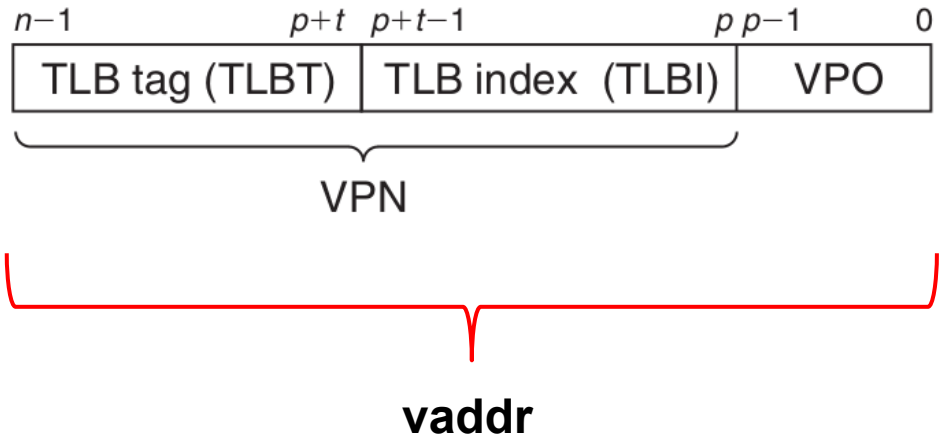


Figure 9.16 **Operational view of a TLB hit and miss.**

TLB: Usage

Internally: Split up VPN into "tag" and "set index" bits (TLBT, TLBI). Use tag, set-index for cache operation.

Figure 9.15
Components of a virtual address that are used to access the TLB.



Tour: Translate Virtual Address

(Based off of example in book: Chapter 9.6.4, pg. 794)

Tour: Translate Virtual Address

Virtual addresses are 14 bits wide

Physical addresses are 12 bits wide

Page table is single-level, with page size of 64 bytes

TLB is four-way set associative w/ 16 total entries.

Question: How many bits are in VPO? VPN?

Answer: Since page size is 64 bytes, we need log2(64) = 6 bits for VPO.
VPN is (14 - 6) = 8 bits.

Question: How many bits are in PPO? PPN?

Answer: Since VPO=PPO, need 6 bits for PPO.
Need (12-6)=6 bits for PPN.

Tour: Translate Virtual Address

Virtual addresses are 14 bits wide

Physical addresses are 12 bits wide

Page table is single-level, with page size of 64 bytes

TLB is four-way set associative w/ 16 total entries.

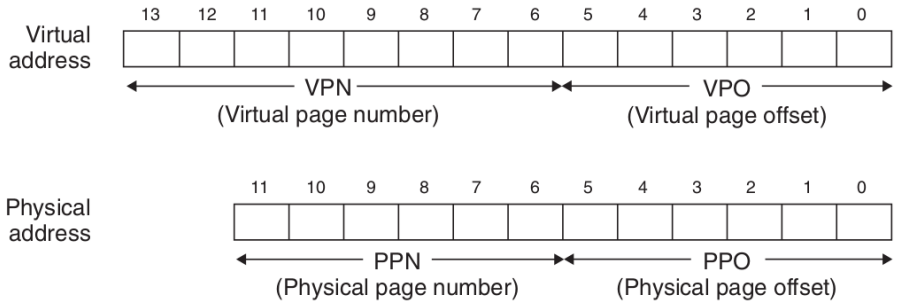


Figure 9.19 **Addressing for small memory system.** Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).

Tour: Translate Virtual Address

Virtual addresses are 14 bits wide
Physical addresses are 12 bits wide
Page table is single-level, with page size of 64 bytes

TLB is four-way set associative w/ 16 total entries.

Question: How many TLB set-index bits do I need?
How many TLB tag bits?

Answer: Since there are four sets, we need log2(4) = 2 set-index bits.
There are (8-2)=6 tag bits.

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

(a) TLB: Four sets, 16 entries, four-way set associative

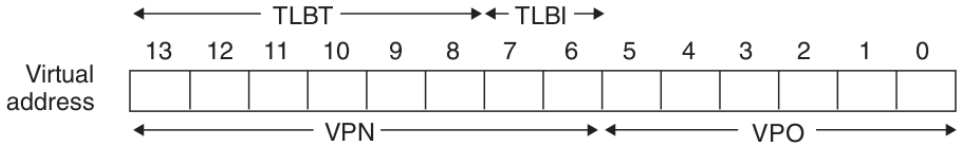
Tour: Translate Virtual Address

Virtual addresses are 14 bits wide
Physical addresses are 12 bits wide
Page table is single-level, with page size of 64 bytes

TLB is four-way set associative w/ 16 total entries.

Question: How many TLB set-index bits do I need?
How many TLB tag bits?

Answer: Since there are four sets, we need log2(4) = 2 set-index bits.
There are (8-2)=6 tag bits.

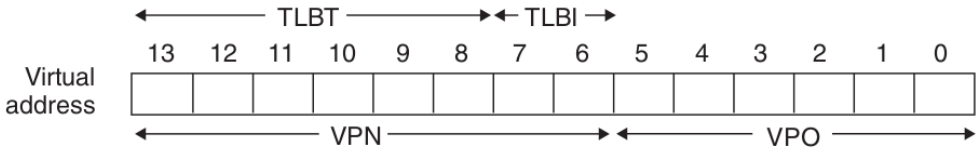


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

(a) TLB: Four sets, 16 entries, four-way set associative

Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **`0x03d4`**. List the steps. What is the result of translating vaddr?



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

(a) TLB: Four sets, 16 entries, four-way set associative

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

(b) Page table: Only the first 16 PTEs are shown

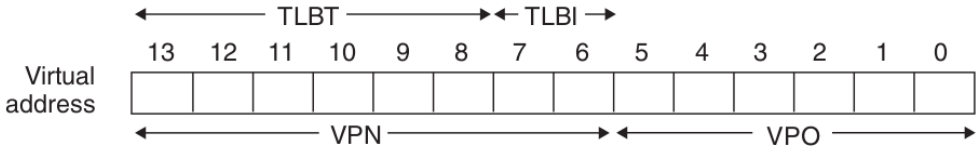
Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **0x03d4**. List the steps. What is the result of translating vaddr?

Answer: TLB -> Hit, PPN=0x0D.
paddr is: 0x354.

Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **`0x023A`**. List the steps. What is the result of translating vaddr?



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

(a) TLB: Four sets, 16 entries, four-way set associative

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

(b) Page table: Only the first 16 PTEs are shown

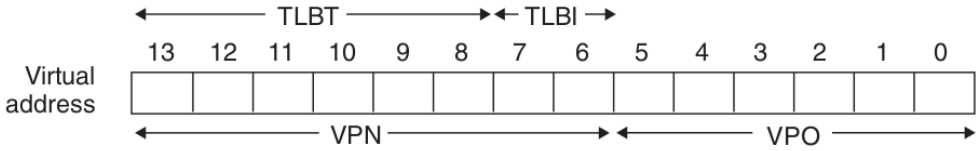
Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **0x023A**. List the steps. What is the result of translating vaddr?

Answer: TLB miss, PageTable(0x08) -> Hit, PPN=0x13.
paddr = 0x4FA.

Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **`0x005F`**. List the steps. What is the result of translating vaddr?



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

(a) TLB: Four sets, 16 entries, four-way set associative

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

(b) Page table: Only the first 16 PTEs are shown

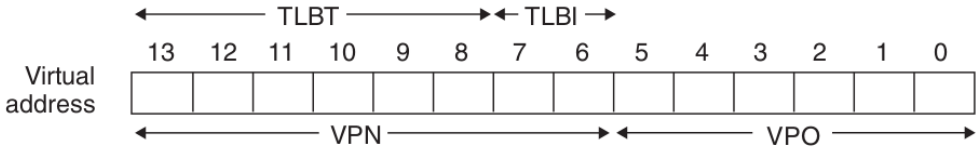
Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **0x005F**. List the steps. What is the result of translating vaddr?

Answer: TLB -> Miss! PageTable(0x01) -> Miss, pagefault!

Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **`0x003A`**. List the steps. What is the result of translating vaddr?



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

(a) TLB: Four sets, 16 entries, four-way set associative

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

(b) Page table: Only the first 16 PTEs are shown

Tour: Translate Virtual Address

Exercise: Suppose CPU requests to load a value from vaddr **0x003A**. List the steps. What is the result of translating vaddr?

Answer: TLB -> Miss (Invalid). PageTable(0x00) -> Hit, PPN=0x28. paddr = 0xA3A

TLB Invalidation

TLB is just another cache.

Question: Under what scenarios do we have to invalidate a cache entry?

Answer:
(1) We do a context switch (P1's vaddr 0x7fffabcd != P2's vaddr 0x7fffabcd).
How to work around this?
(2) When a page fault occurs, and the evicted page resides in the TLB. Since the evicted page no longer is in memory, we must invalidate TLB entry.
(3) Cache replacement. When there is a TLB miss, and a page table hit, then we will replace an older TLB entry with the new mapping.

Context Switch

Recall: A CPU can only execute instructions from one process at a time.

To support multiprocessing, CPU's will periodically switch processes to ensure that each process gets execution time "fairly". Controlled by OS's process scheduler.

Switching from one process to another is called a **context switch**.

Context Switch

Question: Upon a process context switch, what elements of the CPU need to be saved/restored/modified?

Answer:
Need to save/restore: registers, program counter, memory.
Need to invalidate/flush: TLB.

Why do I not need to invalidate Caches (L1,L2,etc.)?
Caches use **physical** addresses, not virtual addresses! P2 will never use the physical address of P1, since addr spaces of P1, P2 are distinct.

TLB: Context Switch

Question: How to avoid having to flush TLB each time CPU performs a process context switch? Pros/Cons?

Answer: Can add a process-ID tag (PID) to each TLB cache entry. Upon a context switch, don't have to invalidate anything! Must modify MMU to also check current PID.
Pros: Fewer TLB misses due to process context switches.
Cons: Makes MMU hardware more complex/expensive.

Nested Page Tables

Suppose we have a single-level page table.

32-bit address space, 4 KiB pages, 4-byte PTE.

Question: How large would the page table need to be to address entire address space (in MiB)?

Answer:
bits PPO = log2(4*2^10) = log2 (2^12) = 12.
Since 32-bit addrspace, there are (32-12)=20 bits in PPN.
Thus, there are 2^20 entries in page table.
Size = (4 bytes) * (2^20) = 4 MiB.

Nested Page Tables

Suppose we have a single-level page table.

64-bit address space, **4 MiB** pages, **9-byte** PTE.

Question: How large would the page table need to be to address entire address space?

Answer:
# bits of PPO = log2(4*2^20) = 22.
# bits PPN = (64-22) = 42.
# entries = 2^42
Size = (9 bytes) * (2^42) = 36 TiB = 36,864 GiB

Ouch! Not feasible to store entire page table in memory!

Nested Page Tables

Solution: Instead of a single flat page table, use nested page tables.

Each PTE points to the start of another page table.

Final page table contains actual PPN's.

Location of first page table is given by register CR3.

Example: Two-level Page Table

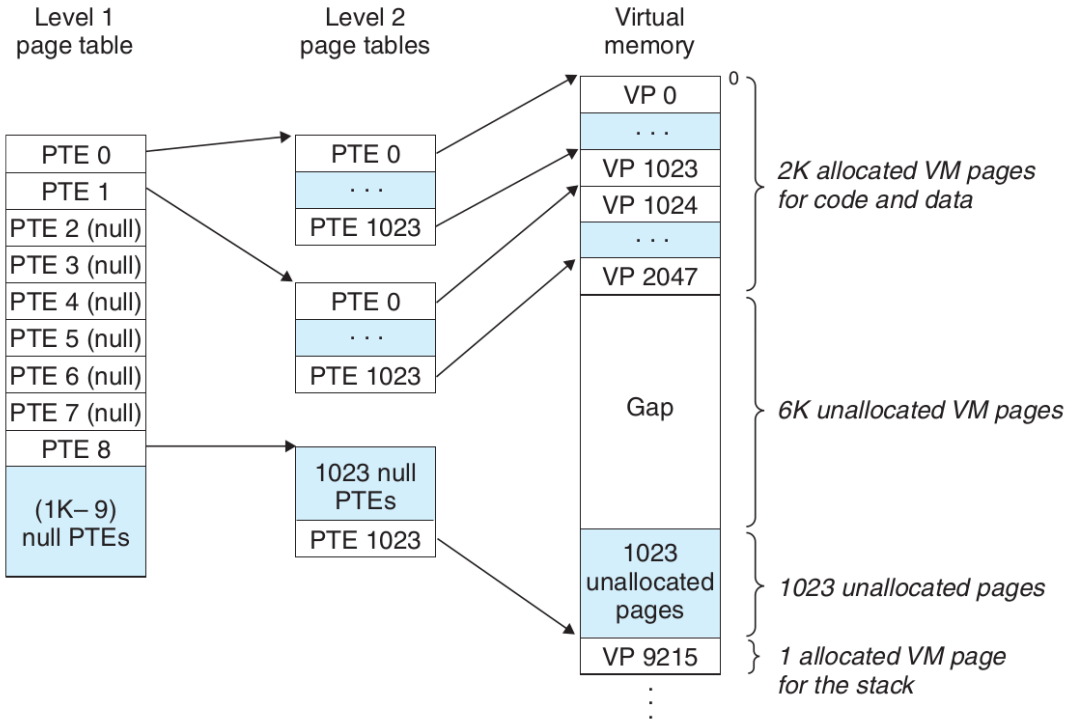


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

Example: Two-level Page Table

Idea: If a program only uses a tiny part of its address space, then most of the PTE's at L1 will be null.
Thus, no need to create PTE's for unused addresses!

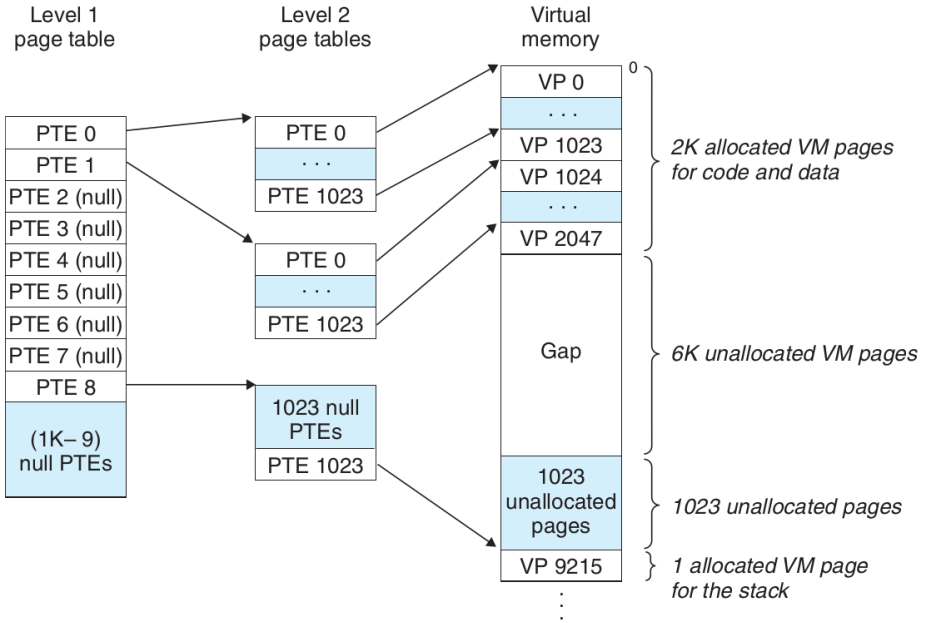


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

Nested Page Tables

Each page of L1 page table references large chunks of address space (ie 4 MiB).

Pages of L2 page table reference smaller chunks (ie 4 KiB).

Similarly for L3, L4, etc.

Nested Page Tables

Only need to keep L1 page table in main memory at all times!

Other levels (L2,L3,L4,etc.) can be paged in/out as necessary (ie stored on disk/restored to DRAM).

Nested Page Tables

Suppose page table has k-levels. Partition VPN into k-sets. Use each VPN_i as the index into the i-th page table.

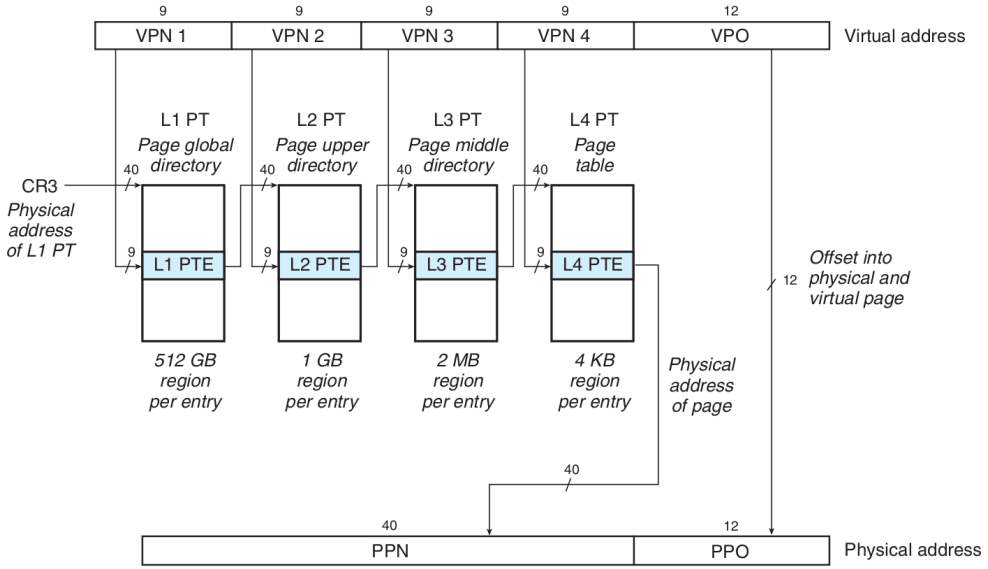


Figure 9.25 **Core i7 page table translation.** Legend: PT: page table, PTE: page table entry, VPN: virtual page number, VPO: virtual page offset, PPN: physical page number, PPO: physical page offset. The Linux names for the four levels of page tables are also shown.

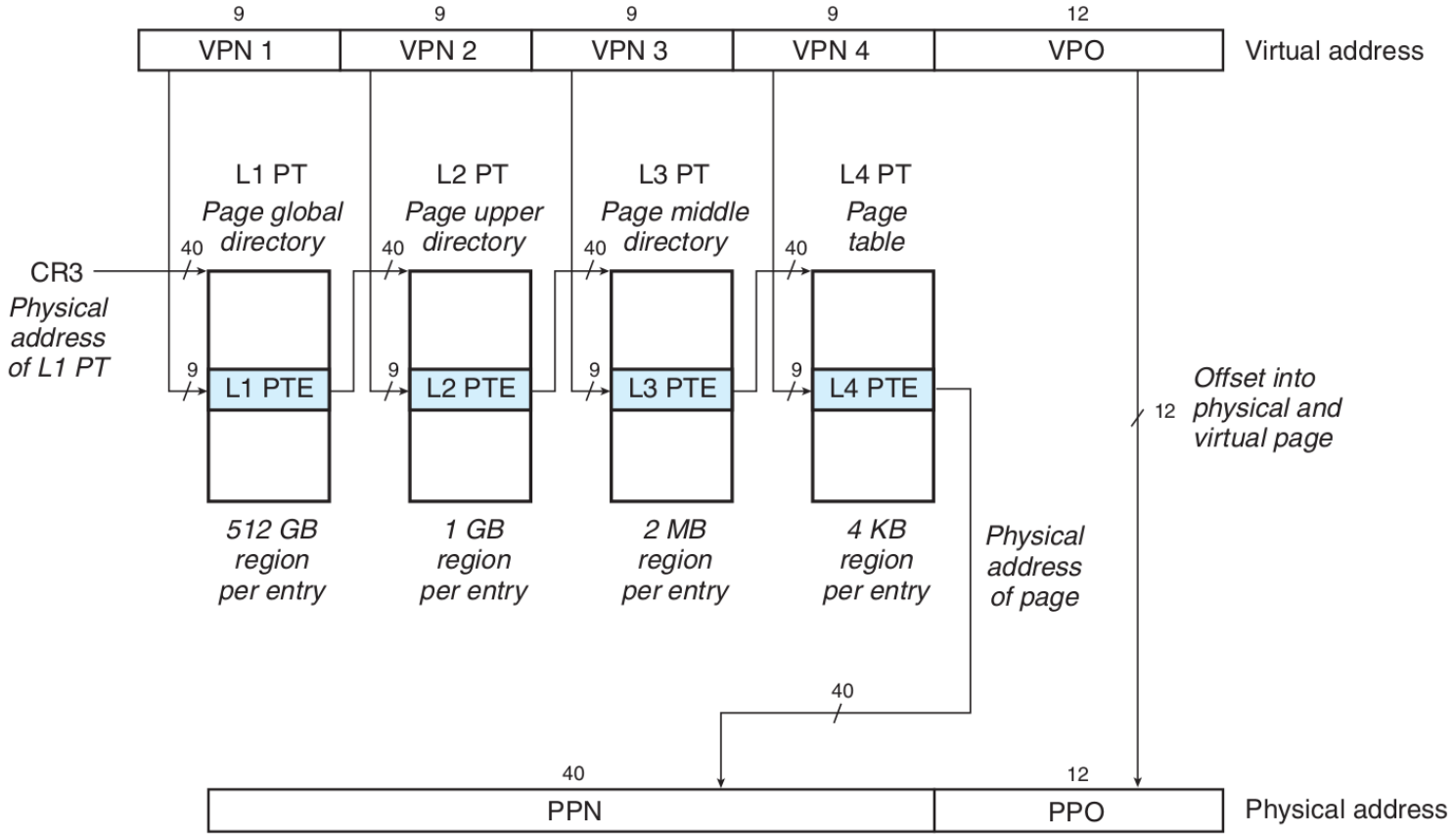


Figure 9.25 **Core i7 page table translation.** Legend: PT: page table, PTE: page table entry, VPN: virtual page number, VPO: virtual page offset, PPN: physical page number, PPO: physical page offset. The Linux names for the four levels of page tables are also shown.

Linking

Compilation pipeline:

```
gcc -S code.c // Compile C -> code.s
gcc -c code.s // Assemble code.s -> code.o
gcc code.o // Link objfile to make executable -> ./a.out
```

Object Files

Aka "Relocatable Object Files".

On unix systems, .o files are in ELF format ("Executable and Linkable Format").

Are binary objects that contain several sections:

header, text, rodata, data, bss, symtable, rel_text, rel_data, debug, line, strtab, footer

Object Files: readelf

Cool trick: Use the `readelf` program to inspect object files!

```
$ readelf -a code.o
...
Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS code.c
...
```

Linking: Demo

Demo: Show how to use CSAPP functions in my C programs.