15-213

"The course that gives CMU its Zip!"

Memory System Case Studies Mar. 20, 2008

Topics

- **P6 address translation**
- **x86-64 extensions**
- **Linux memory management**
- **Linux page fault handling**
- **Memory mapping**

Intel P6

(Bob Colwell's Chip, CMU Alumni)

Internal designation for successor to Pentium

- Which had internal designation P5

Fundamentally different from Pentium

- Out-of-order, superscalar operation

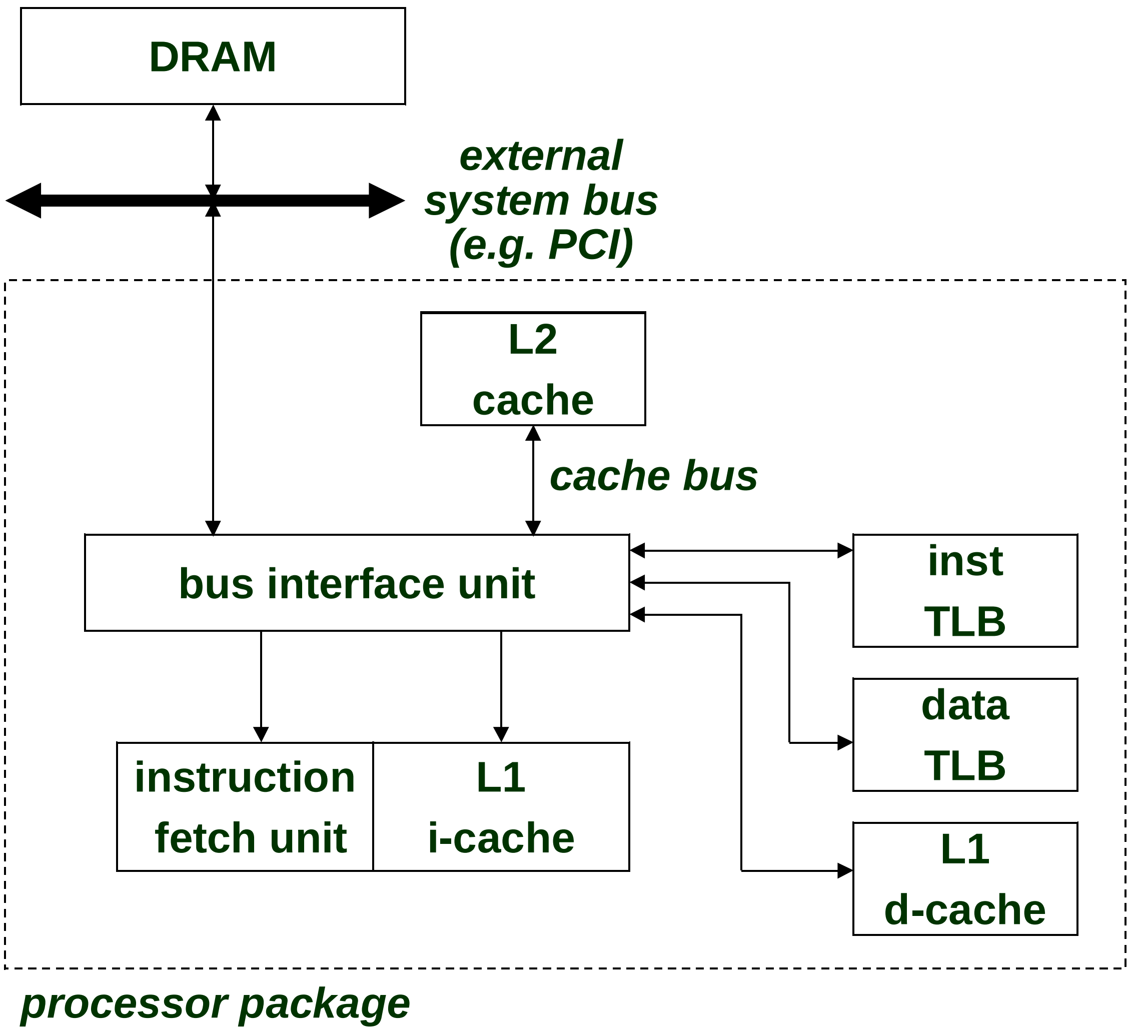
Resulting processors

- Pentium Pro (1996)
- Pentium II (1997)
  - L2 cache on same chip
- Pentium III (1999)
  - The freshwater fish machines

Saltwater fish machines: Pentium 4

- Different operation, but similar memory system
- Abandoned by Intel in 2005 for P6-based Core 2 Duo

P6 Memory System



32 bit address space

4 KB page size

L1, L2, and TLBs
- 4-way set associative

Inst TLB
- 32 entries
- 8 sets

Data TLB
- 64 entries
- 16 sets

L1 i-cache and d-cache
- 16 KB
- 32 B line size
- 128 sets

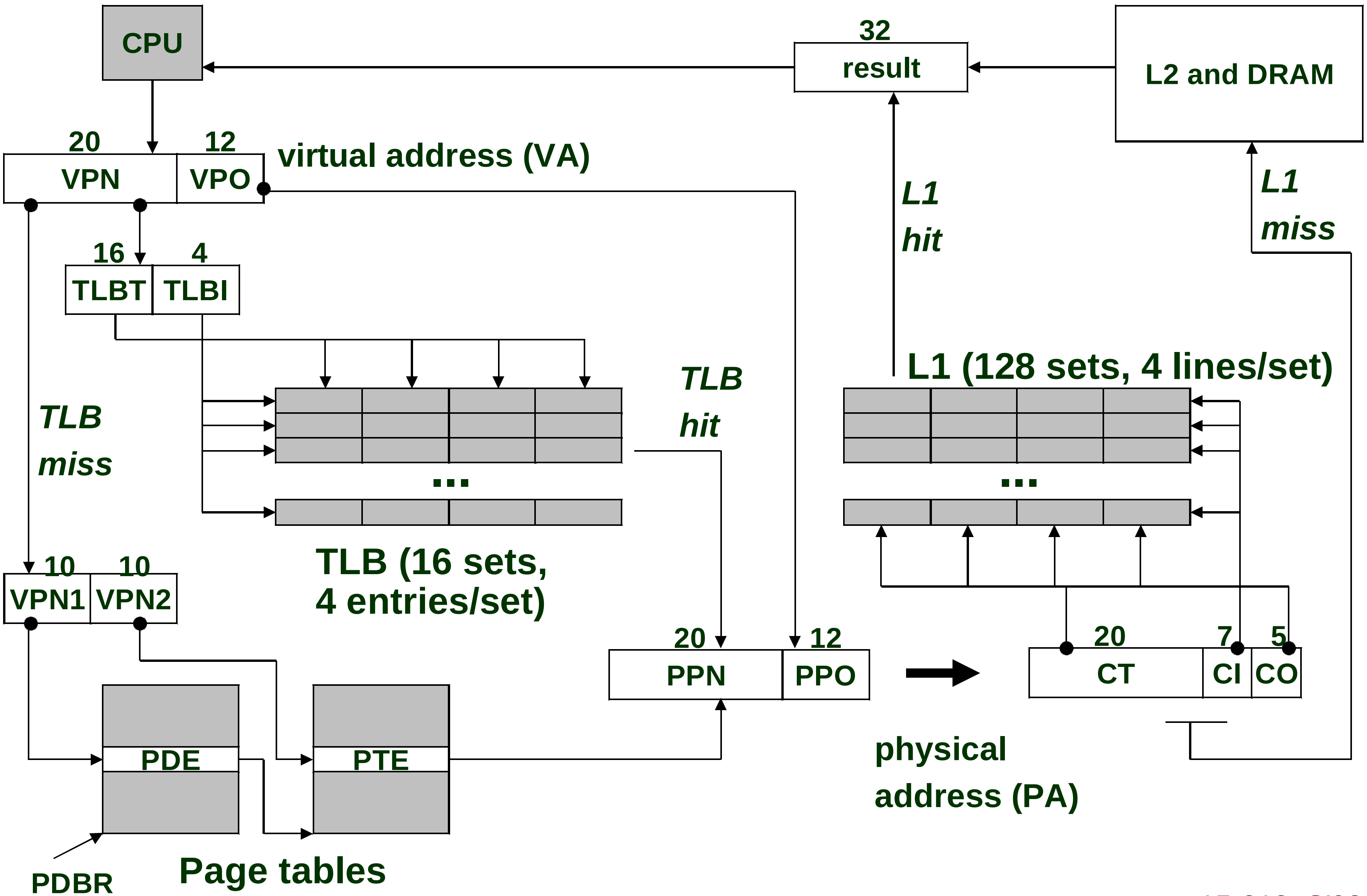
L2 cache
- unified
- 128 KB -- 2 MB

Review of Abbreviations

Symbols:

- **Components of the virtual address (VA)**
  - **TLBI: TLB index**
  - **TLBT: TLB tag**
  - **VPO: virtual page offset**
  - **VPN: virtual page number**
- **Components of the physical address (PA)**
  - **PPO: physical page offset (same as VPO)**
  - **PPN: physical page number**
  - **CO: byte offset within cache line**
  - **CI: cache index**
  - **CT: cache tag**

Overview of P6 Address Translation



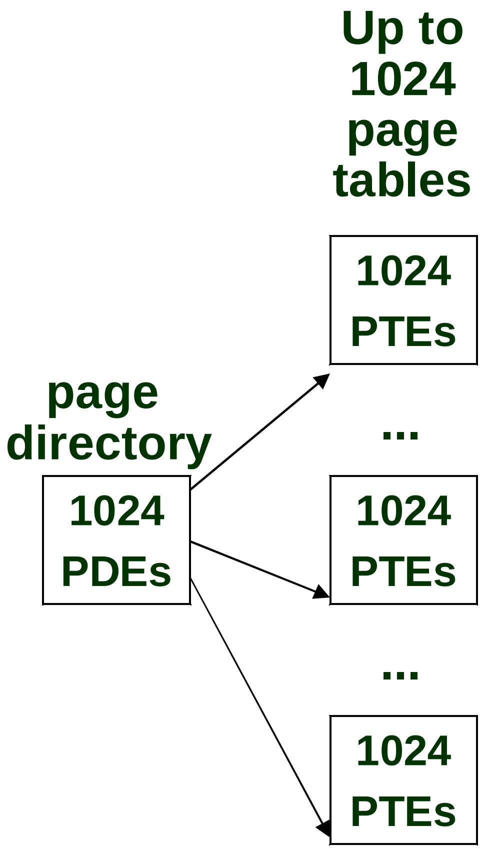
P6 2-level Page Table Structure

Page directory

- 1024 4-byte page directory entries (PDEs) that point to page tables
- One page directory per process.
- Page directory must be in memory when its process is running
- Always pointed to by PDBR

Page tables:

- 1024 4-byte page table entries (PTEs) that point to pages.
- Page tables can be paged in and out.



P6 Page Directory Entry (PDE)

31–12	11–9	8	7	6	5	4	3	2	1	0
Page table physical base addr	Avail	G	PS		A	CD	WT	U/S	R/W	P=1

Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

Avail: These bits available for system programmers

G: global page (don't evict from TLB on task switch)

PS: page size 4K (0) or 4M (1)

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

CD: cache disabled (1) or enabled (0)

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

U/S: user or supervisor mode access

R/W: read-only or read-write access

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

31–1	0
Available for OS (page table location in secondary storage)	P=0

P6 Page Table Entry (PTE)

31 – 12	11 – 9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

<u>Page base address</u>: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

<u>Avail</u>: available for system programmers

<u>G</u>: global page (don't evict from TLB on task switch)

<u>D</u>: dirty (set by MMU on writes)

<u>A</u>: accessed (set by MMU on reads and writes)

<u>CD</u>: cache disabled or enabled

<u>WT</u>: write-through or write-back cache policy for this page

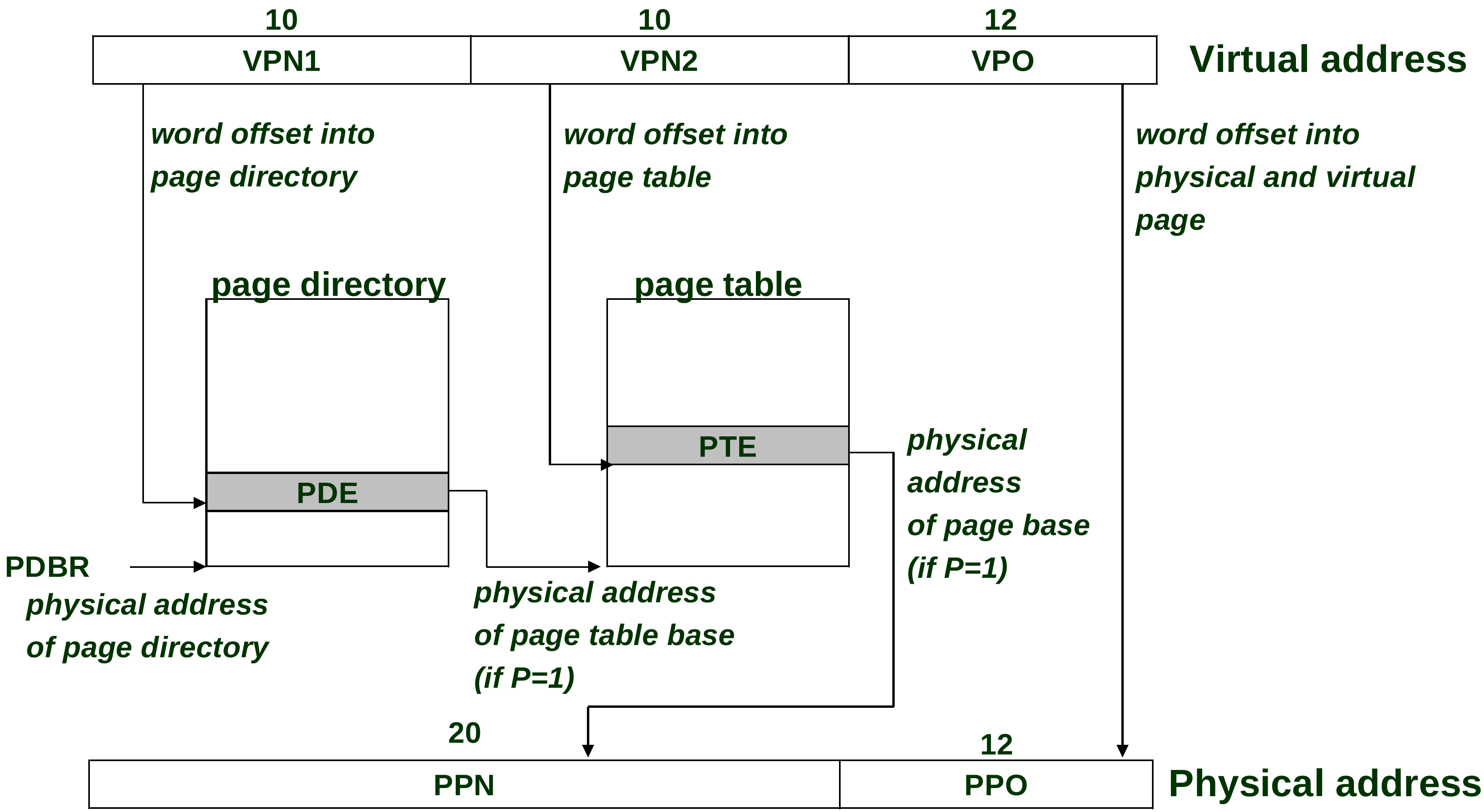
<u>U/S</u>: user/supervisor

<u>R/W</u>: read/write

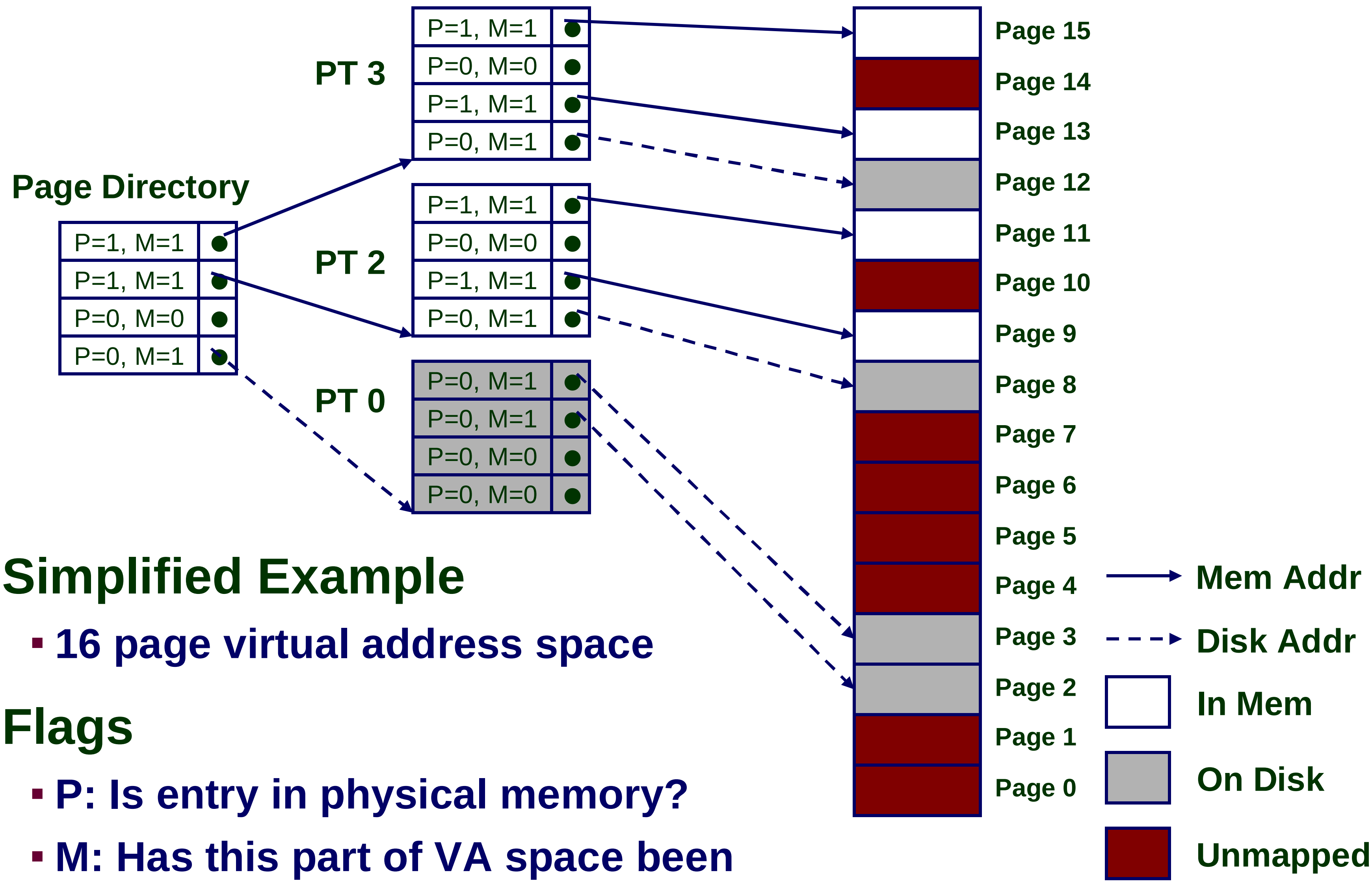
<u>P</u>: page is present in physical memory (1) or not (0)

31 – 1	0
Available for OS (page location in secondary storage)	P=0

How P6 Page Tables Map Virtual Addresses to Physical Ones



Representation of VM Address Space



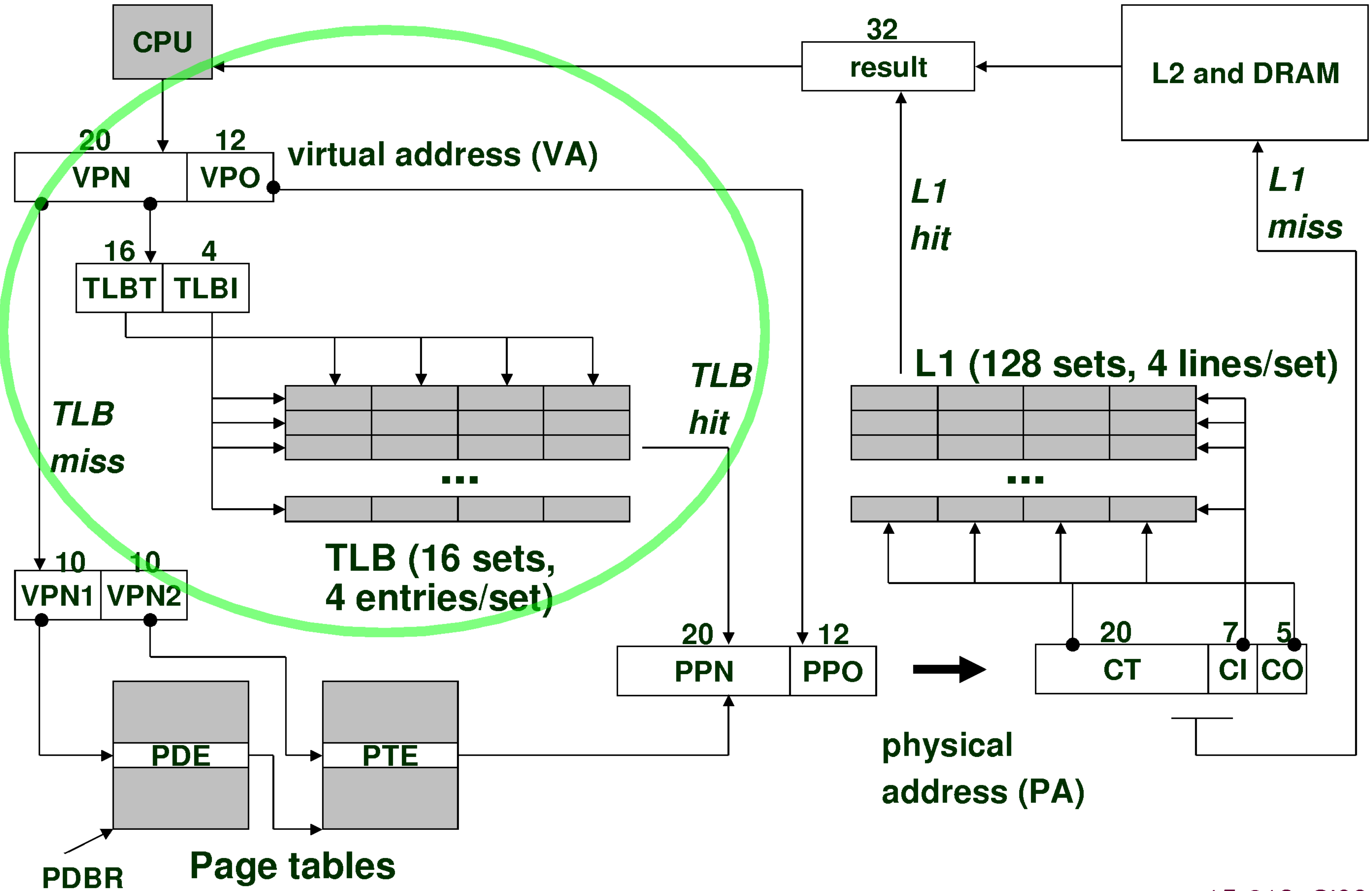
Simplified Example

- 16 page virtual address space

Flags

- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?

P6 TLB Translation



P6 TLB

TLB entry (not all documented, so this is speculative):

20	16	1	1	1	1	1
PPN	Tag	V	G	S	W	D

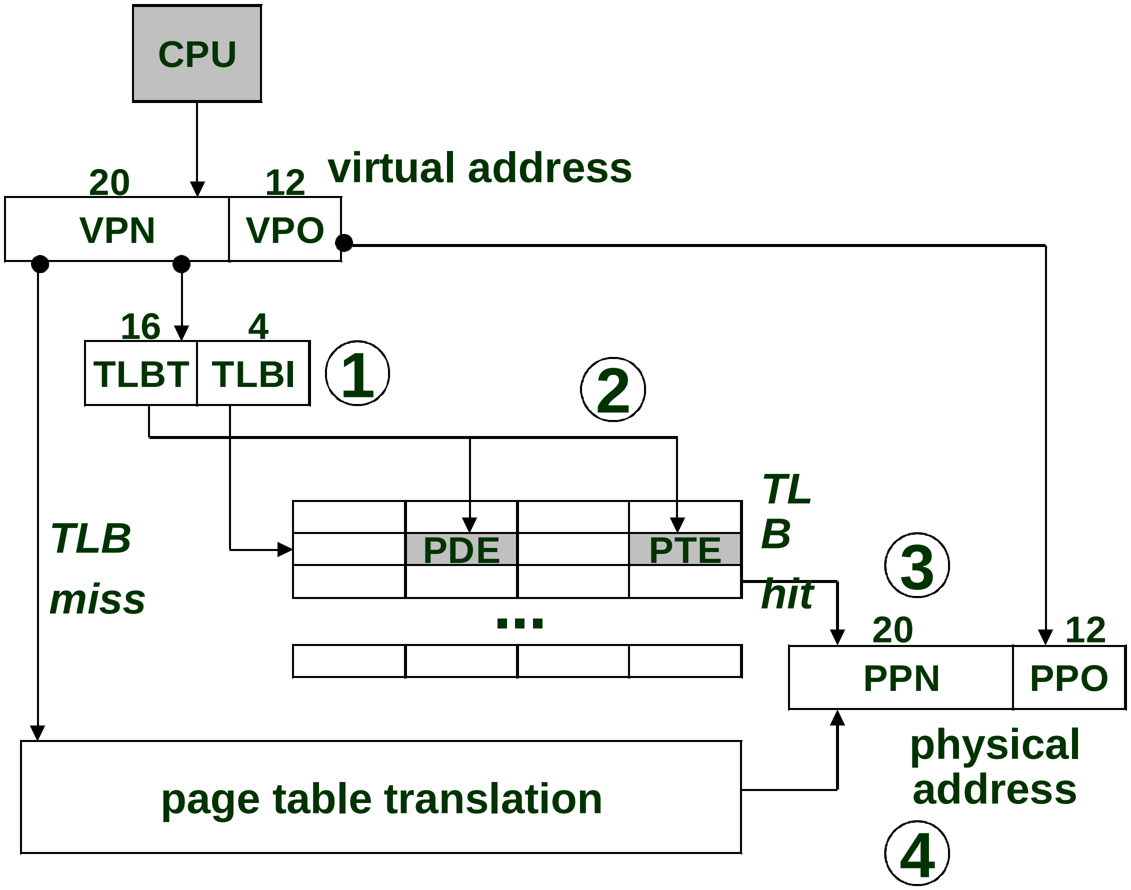
- V: indicates a valid (1) or invalid (0) TLB entry
- Tag: disambiguates entries cached in the same set
- PPN: translation of the address indicated by index & tag
- G: page is “global” according to PDE, PTE
- S: page is “supervisor-only” according to PDE, PTE
- W: page is writable according to PDE, PTE
- D: PTE has already been marked “dirty” (once is enough)

Structure of the data TLB:

- 16 sets, 4 entries/set

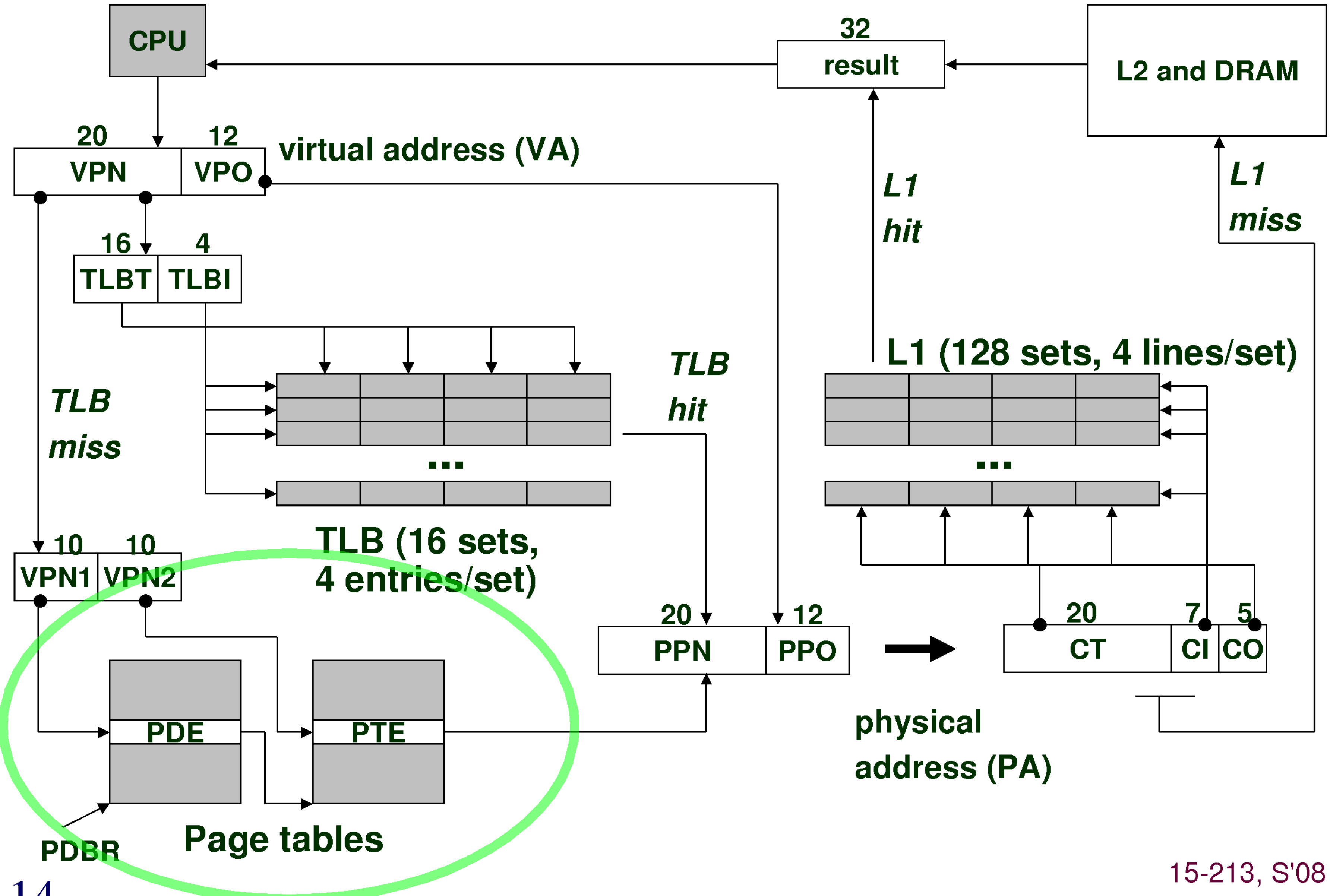
entry	entry	entry	entry	set 0
entry	entry	entry	entry	set 1
...				
entry	entry	entry	entry	set 15

Translating with the P6 TLB

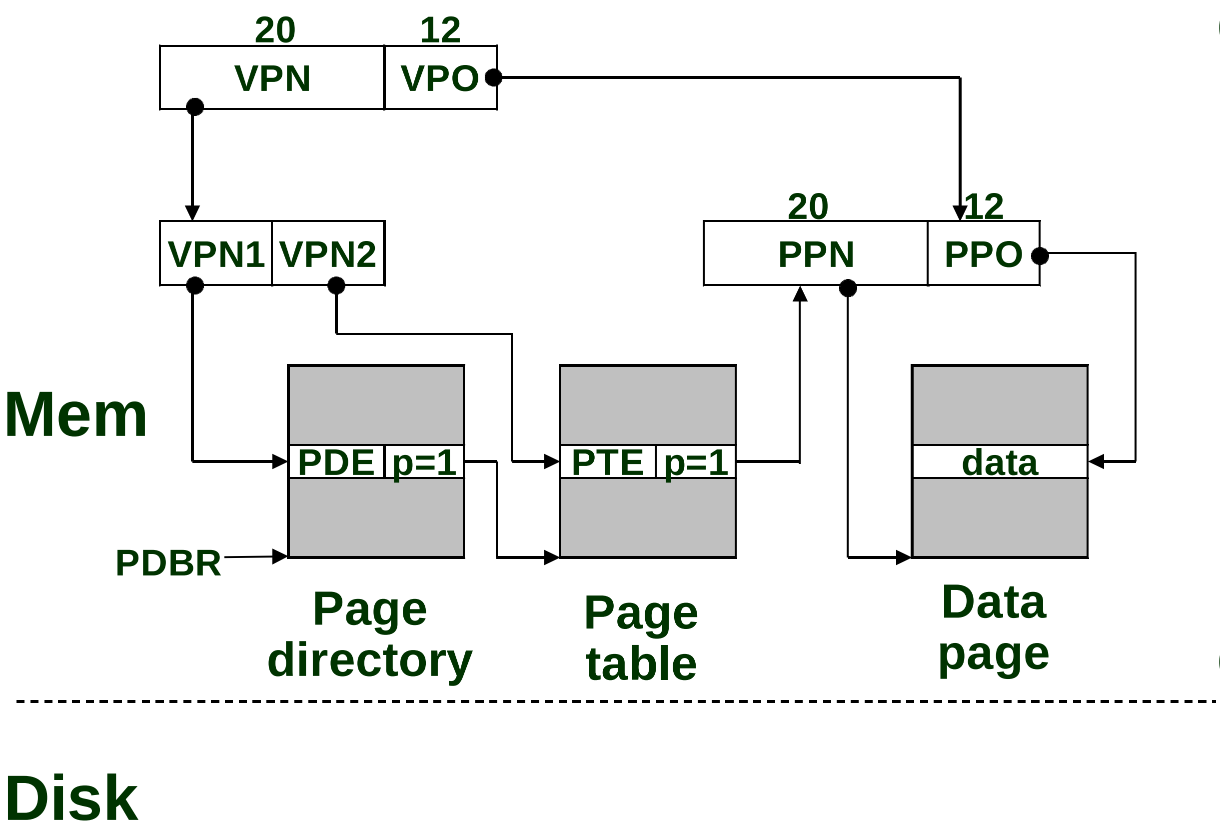


1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
3. Yes: then check permissions, build physical address.
4. No: then read PTE (and PDE if not cached) from memory and build physical address.

P6 Page Table Translation



Translating with the P6 Page Tables (case 1/1)



Case 1/1: page table and page present.

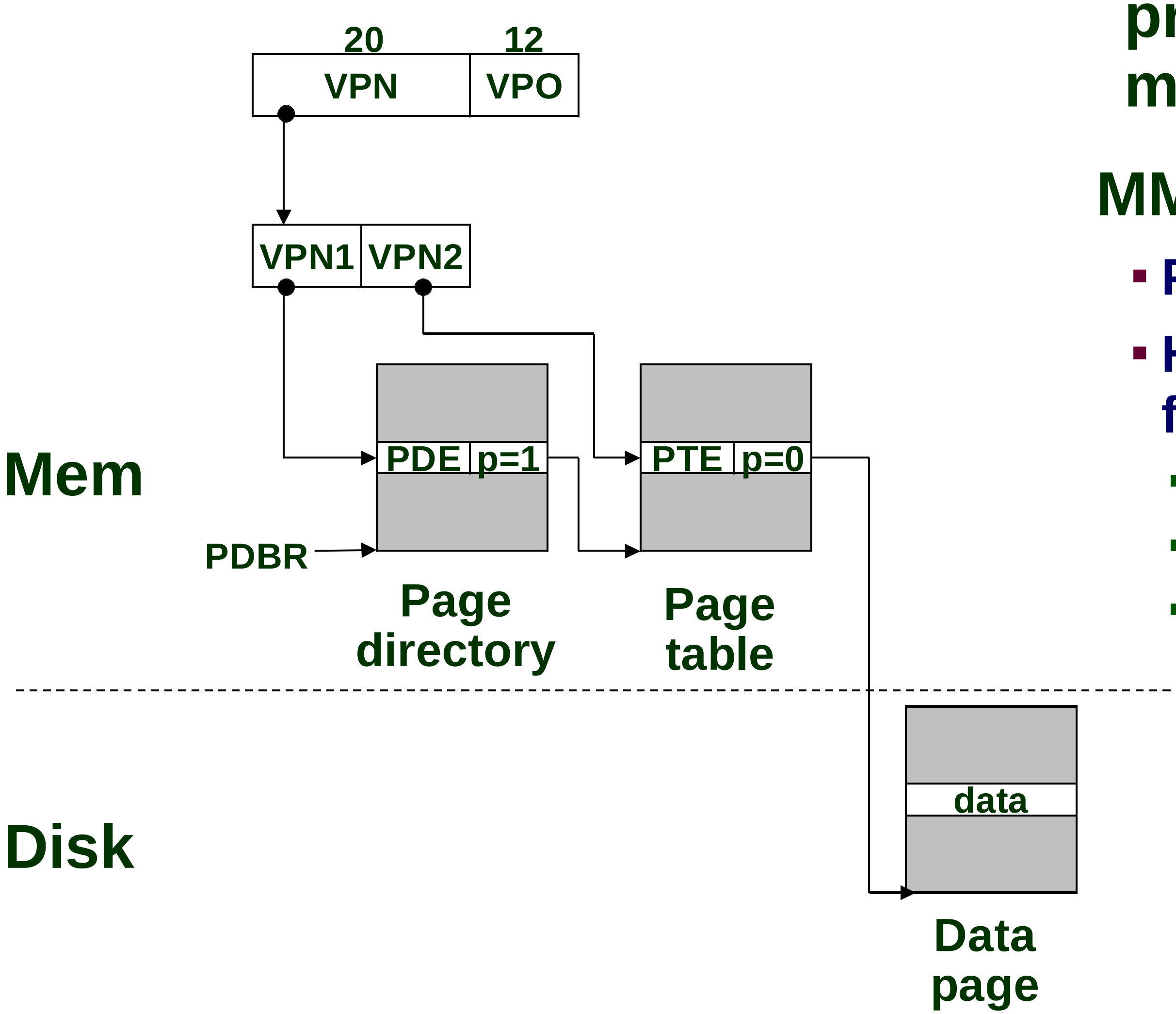
MMU Action:

- MMU builds physical address and fetches data word.

OS action

- None

Translating with the P6 Page Tables (case 1/0)

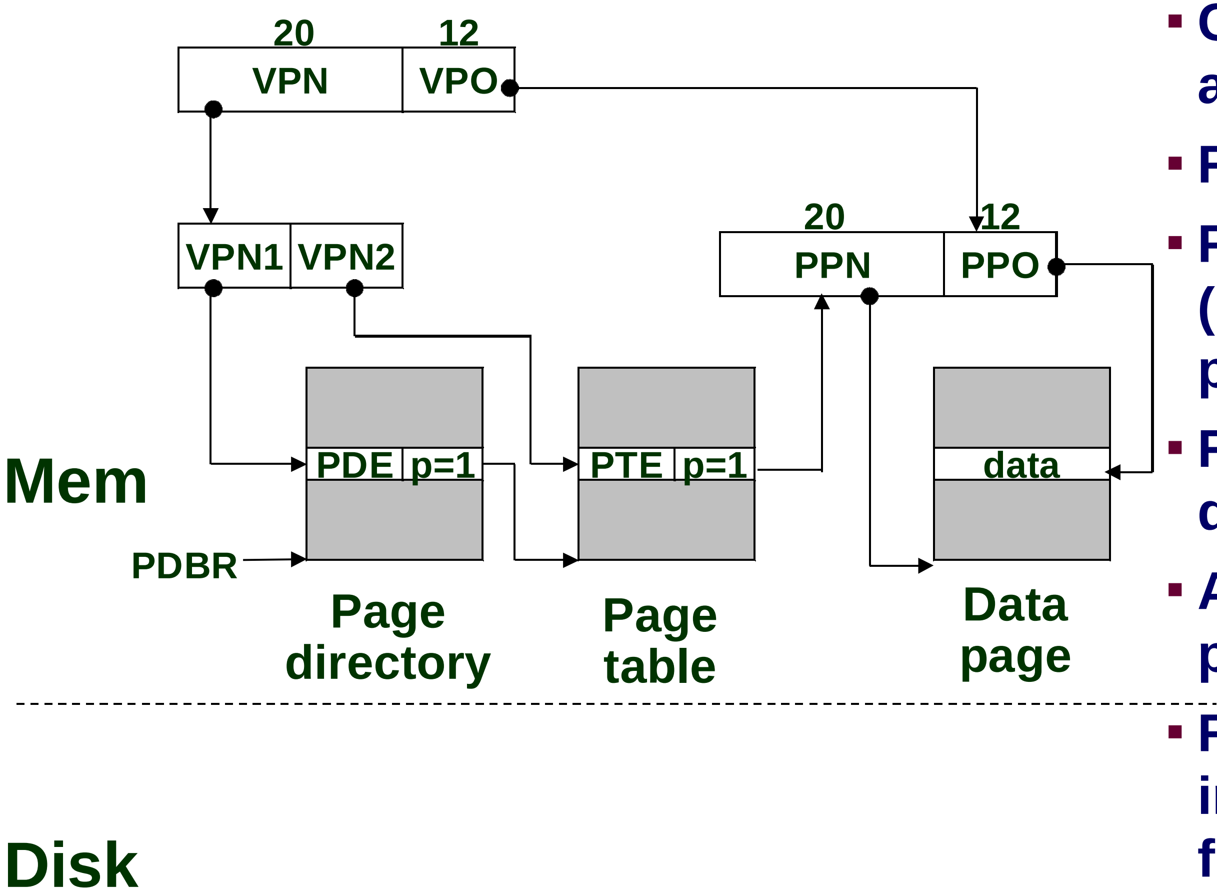


Case 1/0: page table present but page missing.

MMU Action:

- Page fault exception
- Handler receives the following args:
  - %EIP that caused fault
  - VA that caused fault
  - Fault caused by non-present page or page-level protection violation
    - Read/write
    - User/supervisor

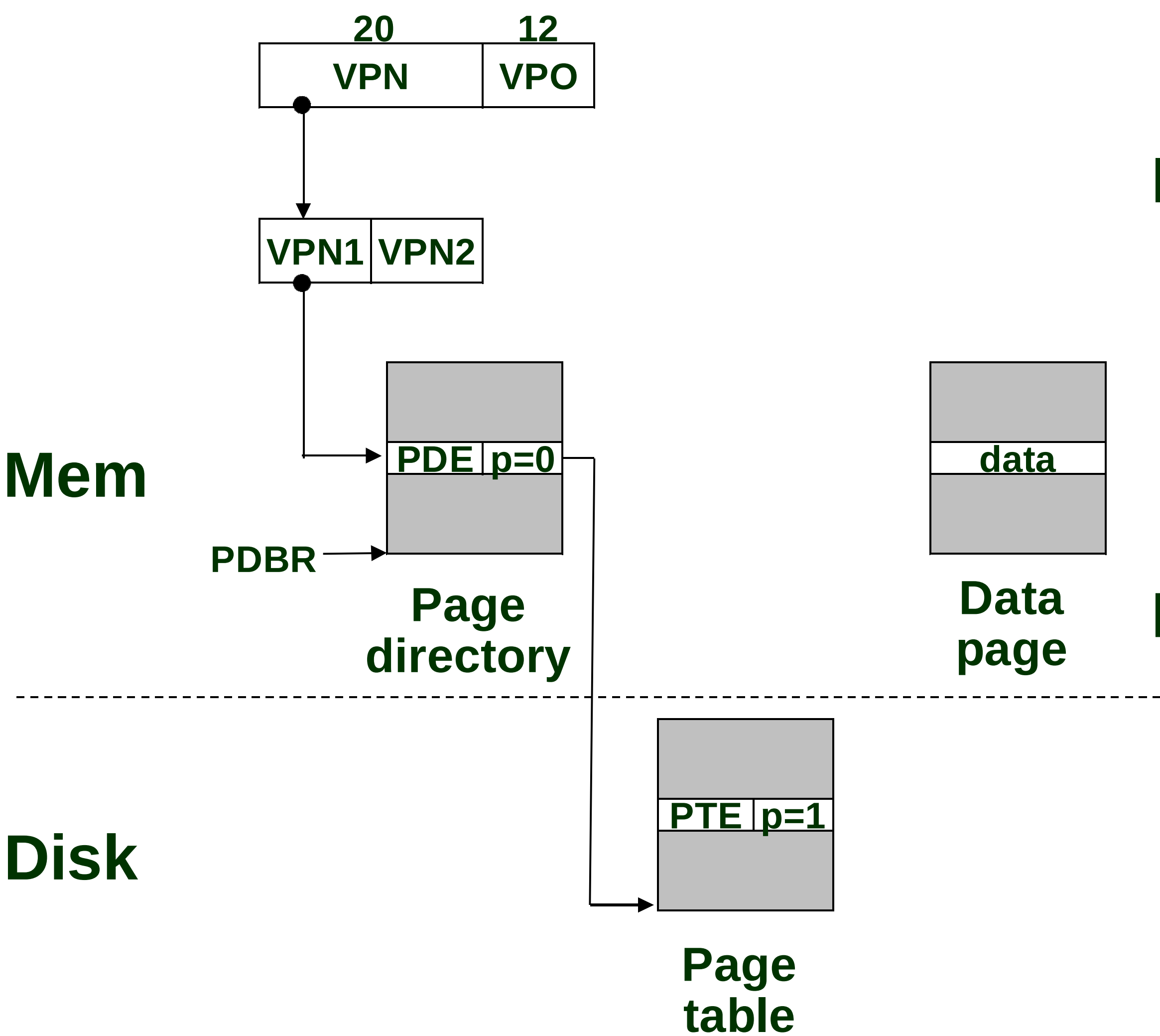
Translating with the P6 Page Tables (case 1/0, cont)



OS Action:

- Check for a legal virtual address.
- Read PTE through PDE.
- Find free physical page (swapping out current page if necessary)
- Read virtual page from disk into physical page
- Adjust PTE to point to physical page, set p=1
- Restart faulting instruction by returning from exception handler.

Translating with the P6 Page Tables (case 0/1)



Case 0/1: page table missing but page present.

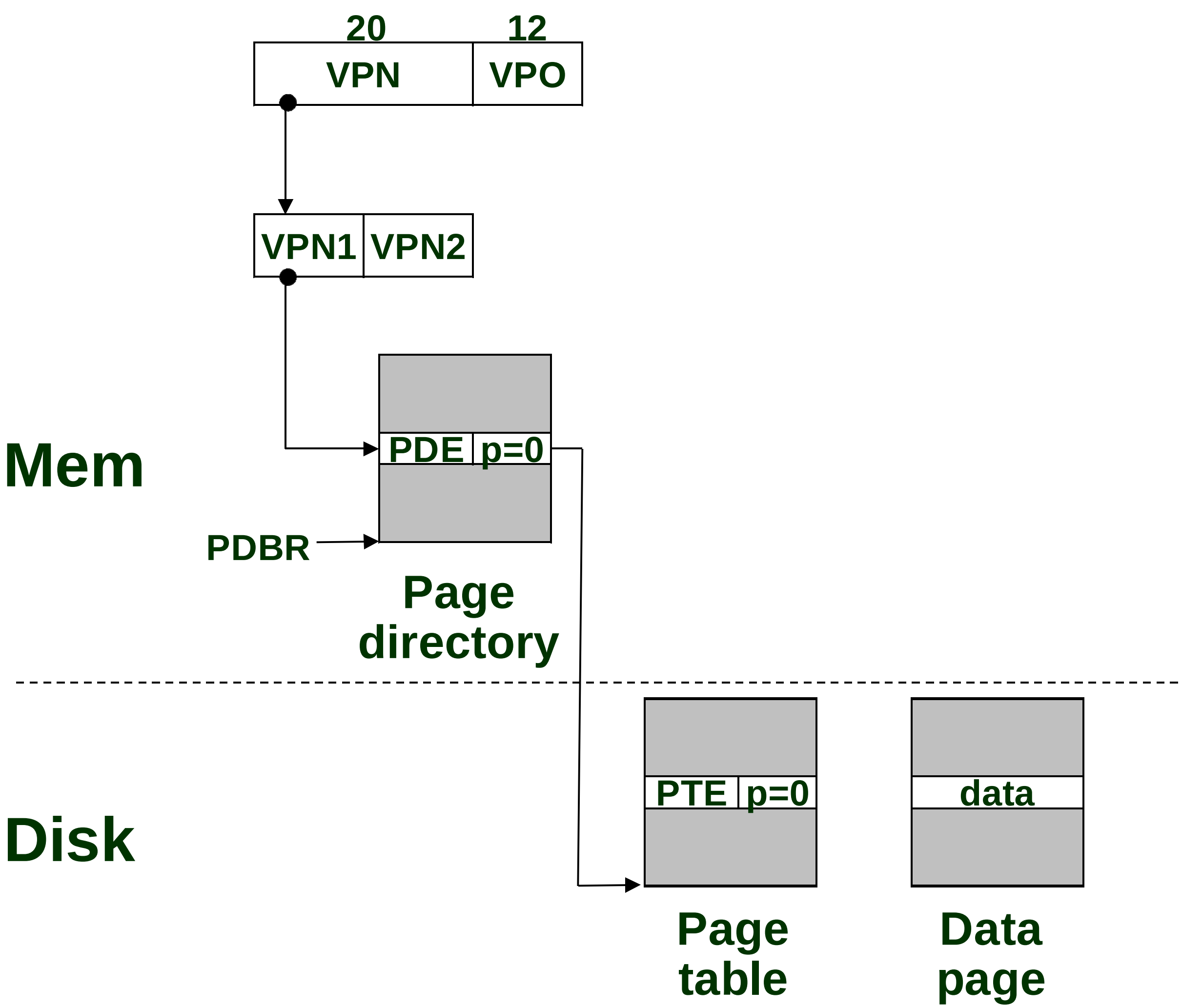
Introduces consistency issue.

- Potentially every page-out requires update of disk page table.

Linux disallows this

- If a page table is swapped out, then swap out its data pages too.

Translating with the P6 Page Tables (case 0/0)

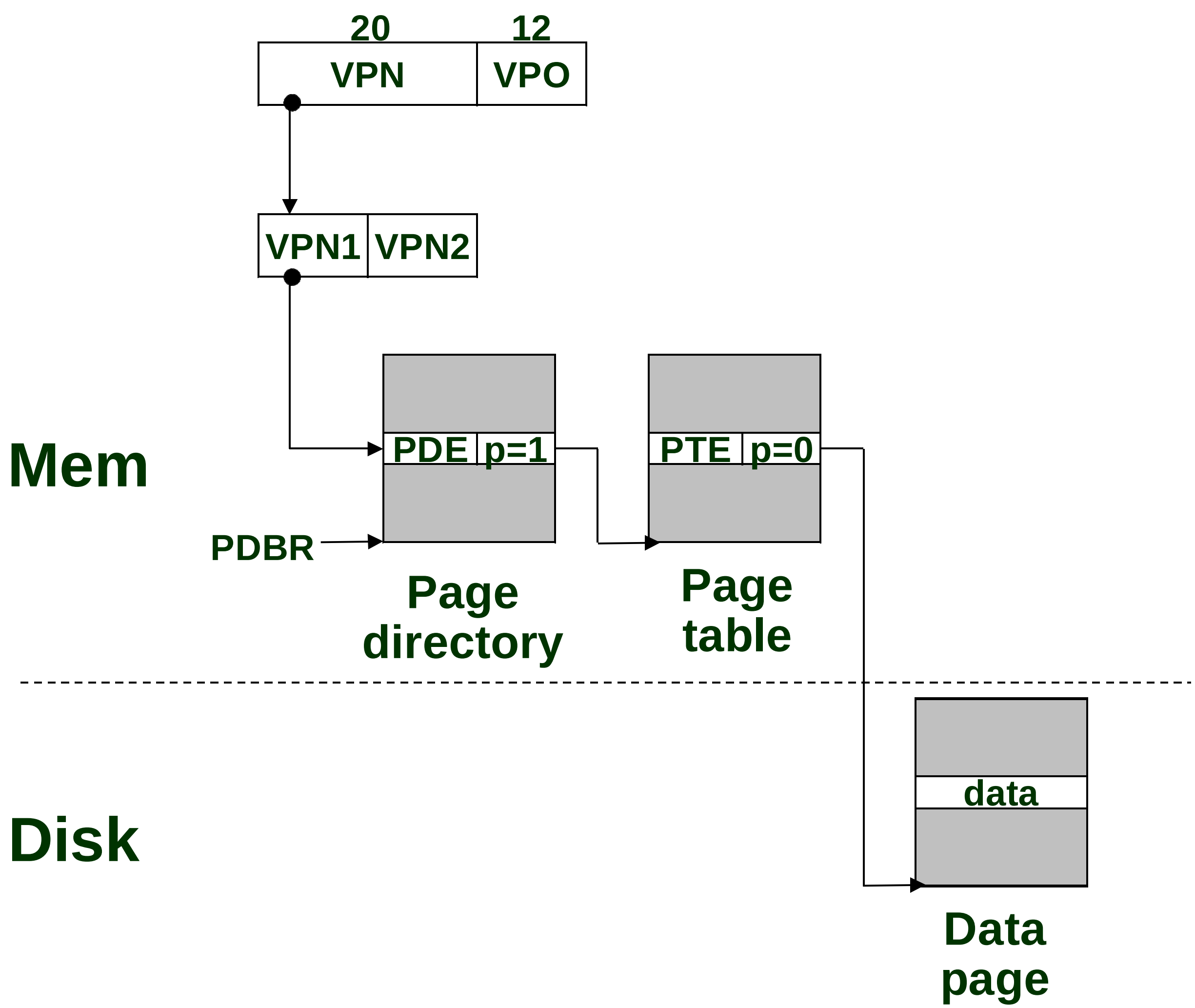


Case 0/0: page table and page missing.

MMU Action:

- Page fault exception

Translating with the P6 Page Tables (case 0/0, cont)



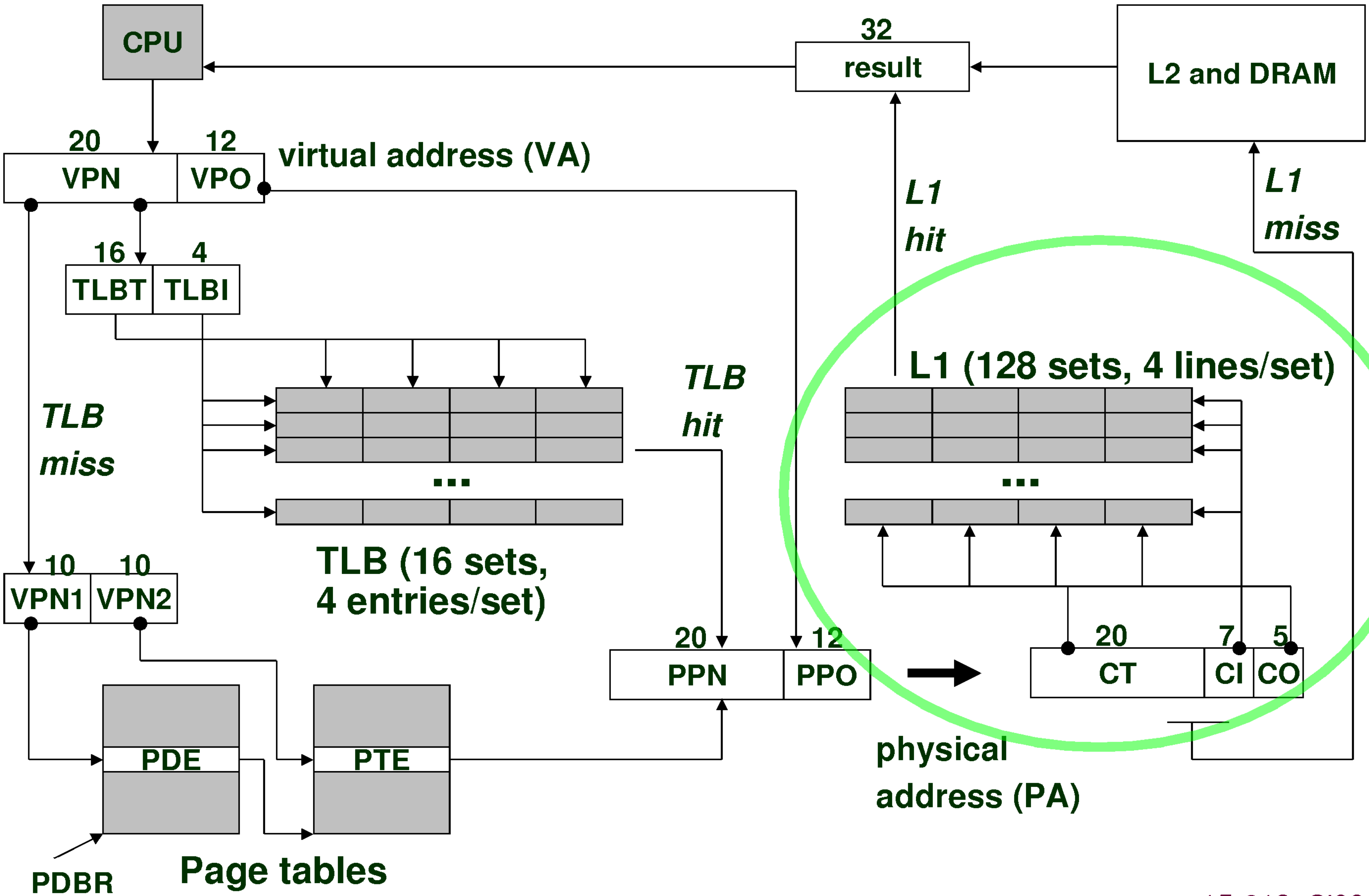
OS action:

- Swap in page table.
- Restart faulting instruction by returning from handler.

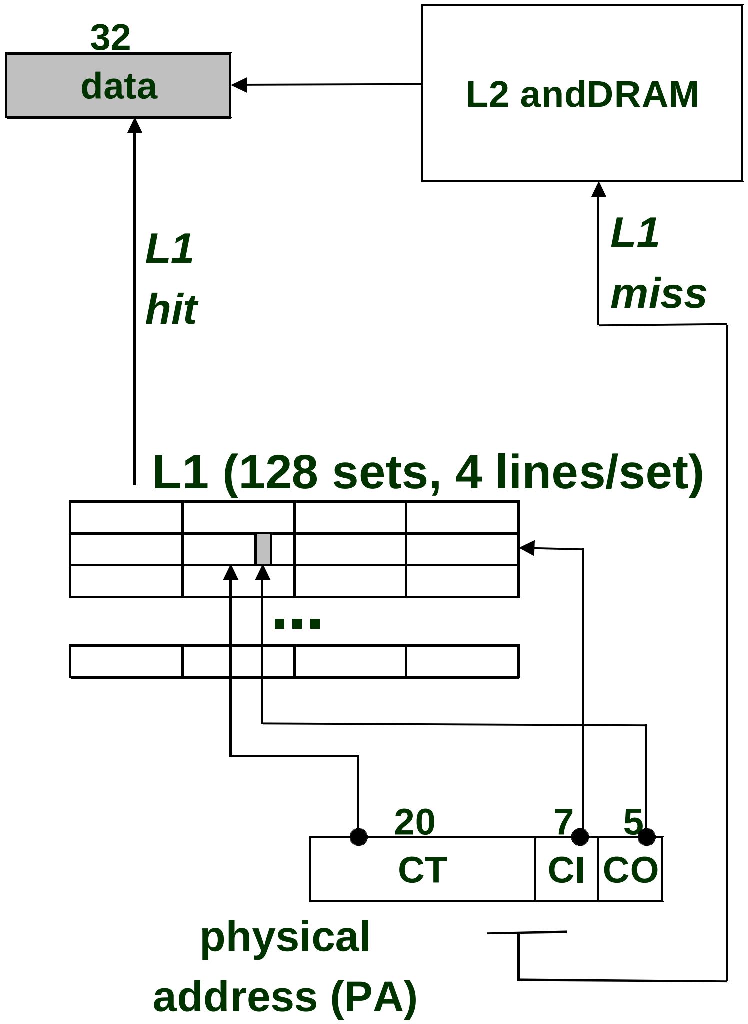
Like case 0/1 from here on.

- Two disk reads (8 sectors each) will probably require >15 ms

P6 L1 Cache Access



L1 Cache Access



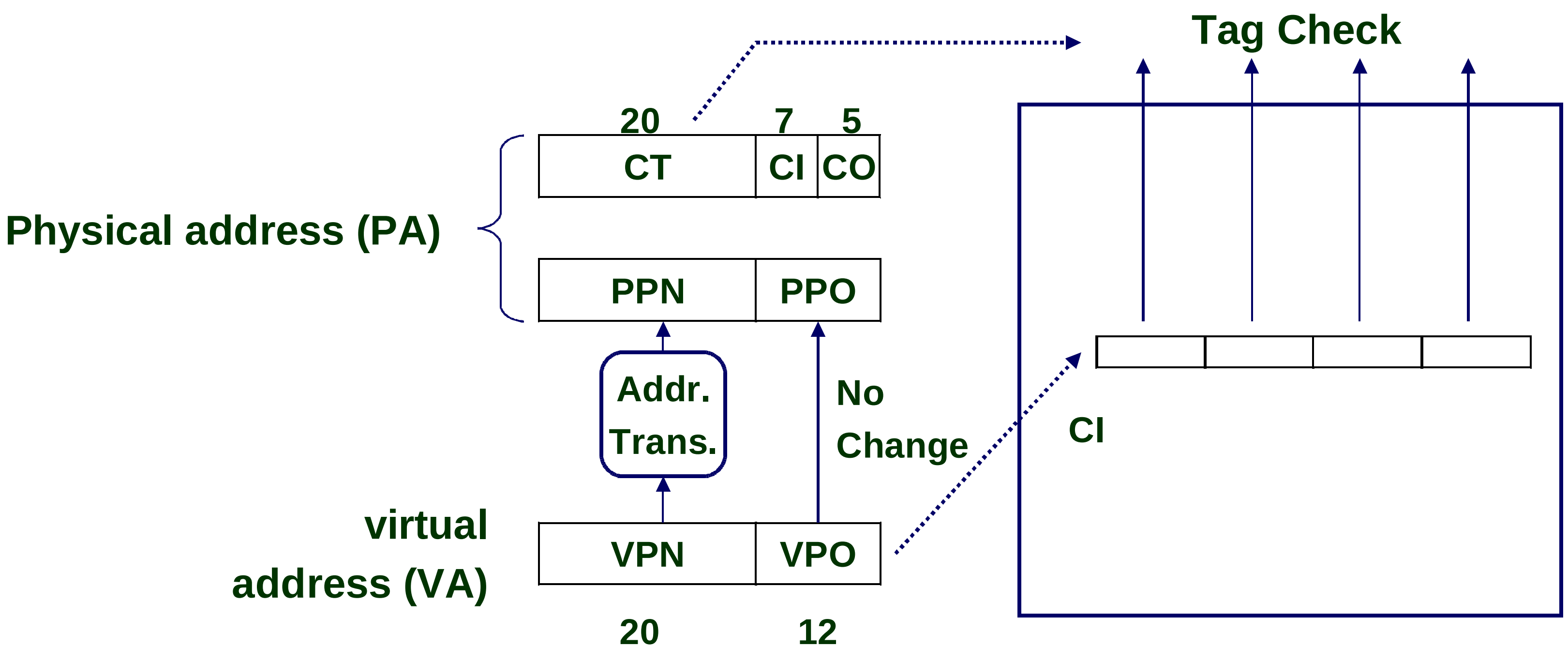
Partition physical address into CO, CI, and CT.

Use CT to determine if line containing word at address PA is cached in set CI.

If no: check L2.

If yes: extract word at byte offset CO and return to processor.

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

x86-64 Paging

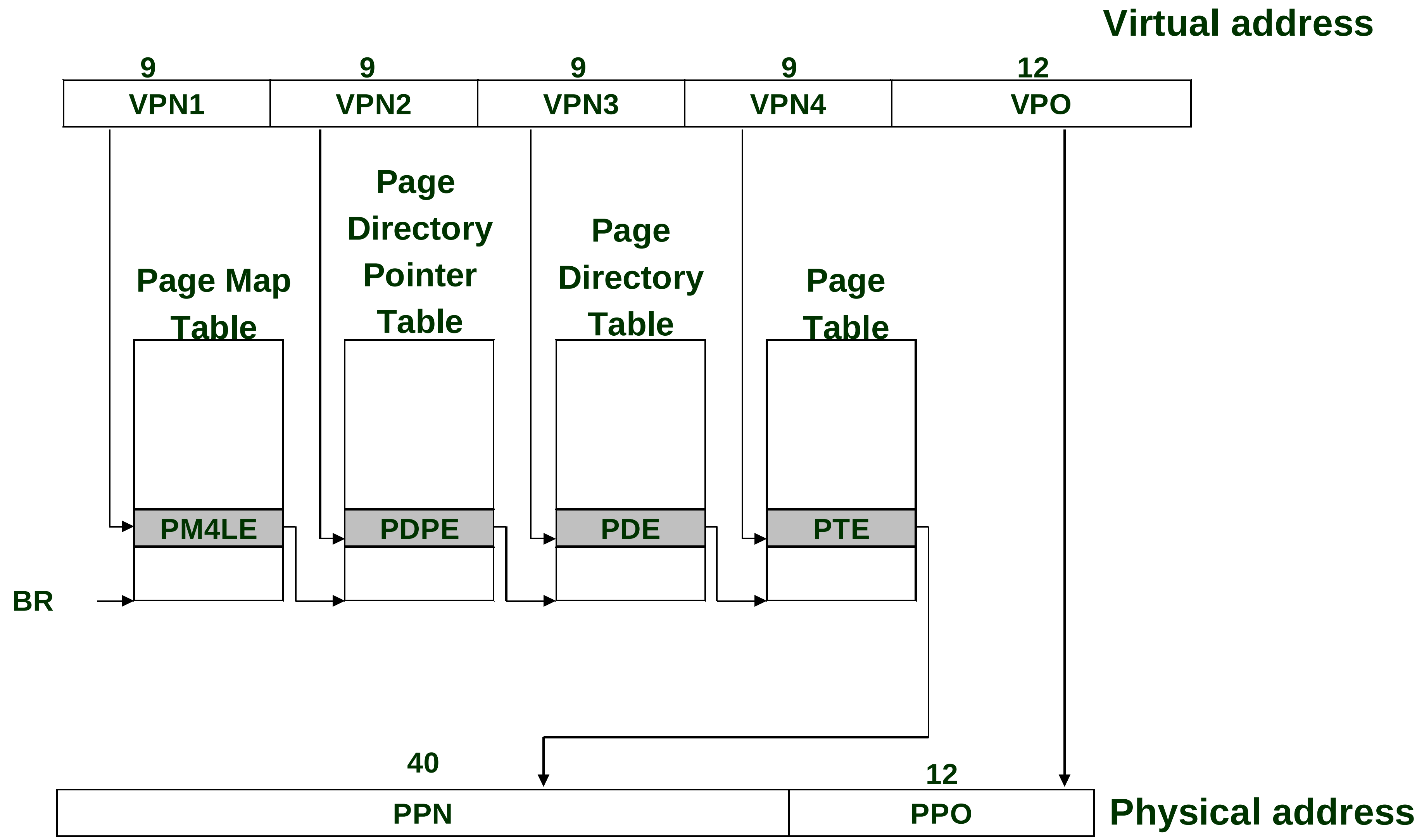
Origin

- AMD's way of extending x86 to 64-bit instruction set
- Intel has followed with "EM64T"

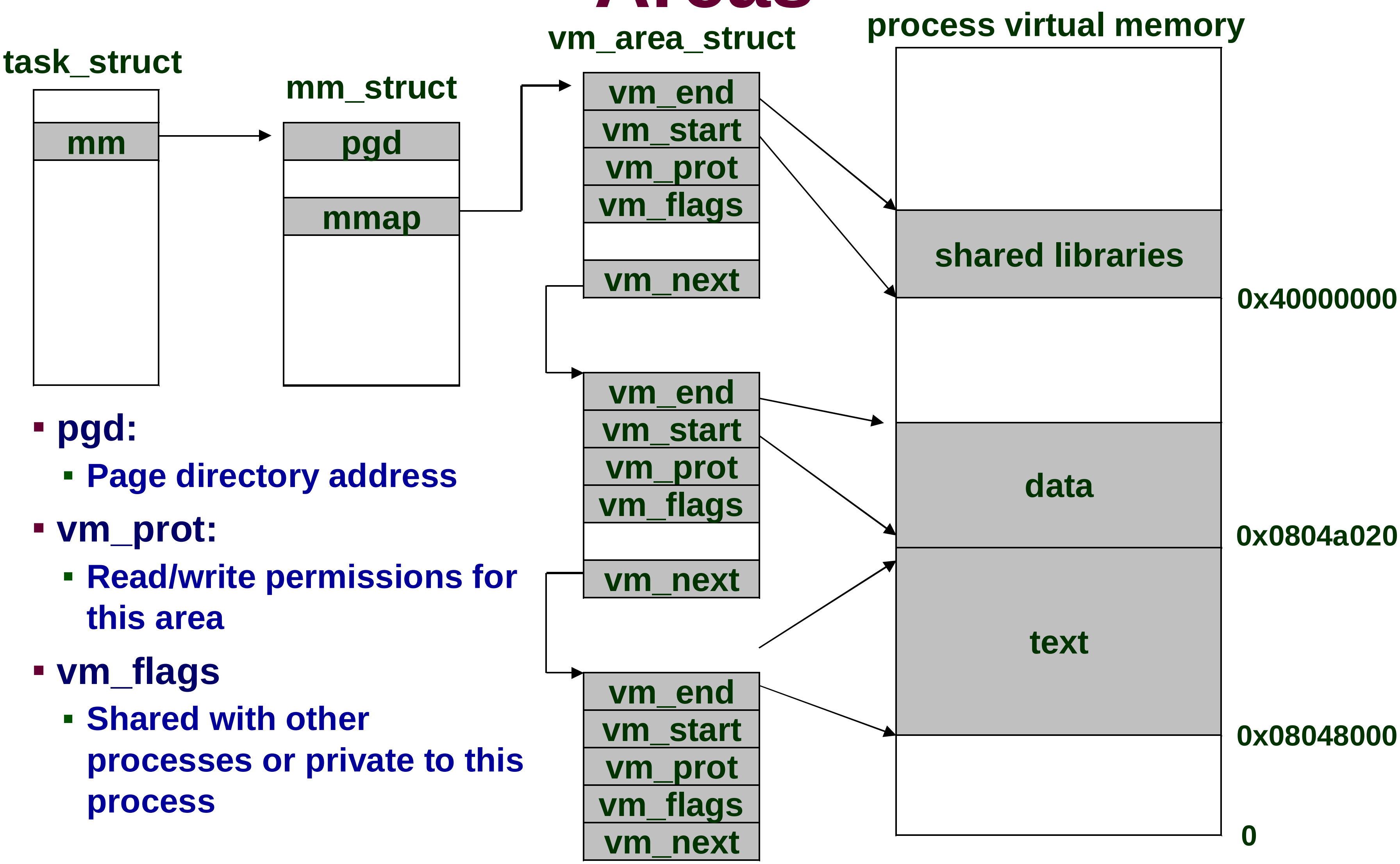
Requirements

- 48-bit virtual address
  - 256 terabytes (TB)
  - Not yet ready for full 64 bits
    - Nobody can buy that much DRAM yet
    - Mapping tables would be huge
    - Multi-level array map may not be the right data structure
- 52-bit physical address
  - Requires 64-bit table entries
- Keep traditional x86 4KB page size
  - (4096 bytes per PT) / (8 bytes per PTE) = only 512 entries per page

x86-64 Paging

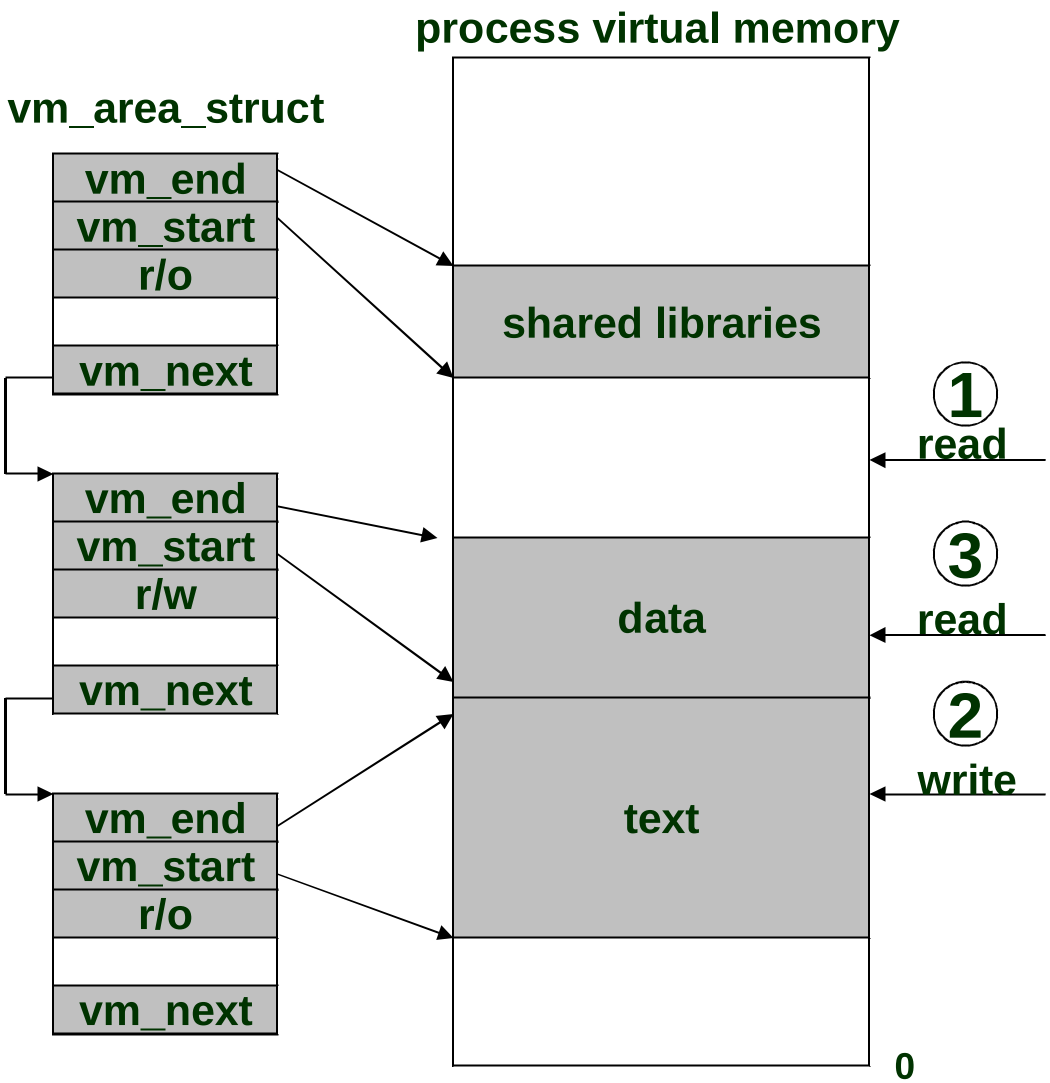


Linux Organizes VM as Collection of "Areas"



- **pgd:**
  - Page directory address
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Shared with other processes or private to this process

Linux Page Fault Handling



Is the VA legal?

- i.e., Is it in an area defined by a vm_area_struct?
- If not (#1), then signal segmentation violation

Is the operation legal?

- i.e., Can the process read/write this area?
- If not (#2), then signal protection violation

Otherwise

- Valid address (#3): handle fault

Memory Mapping

Creation of new VM *area* done via "memory mapping"

- **Create new vm_area_struct and page tables for area**
- **Area can be backed by (i.e., get its initial values from) :**
  - **Regular file on disk (e.g., an executable object file)**
    - **Initial page bytes come from a section of a file**
  - **Nothing (e.g., BSS)**
    - **First fault will allocate a physical page full of 0's**
    - **Once the page is written to (dirtied), it is like any other page**
- **Dirty pages are swapped back and forth between a special swap file.**

<u>Key point</u>: no virtual pages are copied into physical memory until they are referenced!

- **Known as "demand paging"**
- **Crucial for time and space efficiency**

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

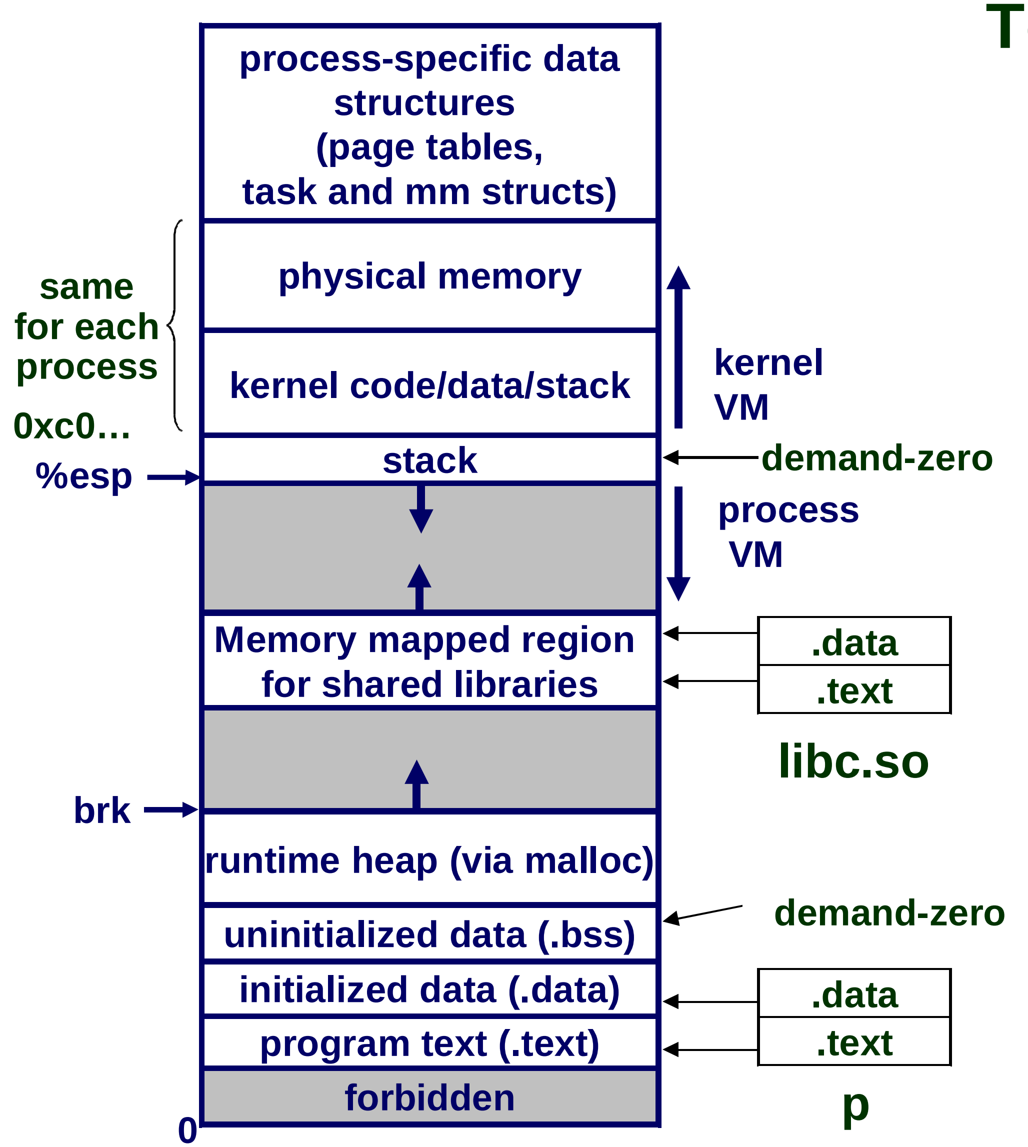
- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
  - `start`: may be 0 for "pick an address"
  - `prot`: MAP_READ, MAP_WRITE
  - `flags`: MAP_PRIVATE, MAP_SHARED
- Return a pointer to start of mapped area (may not be `start`)
- Example: fast file-copy
  - Useful for applications like Web servers that need to quickly copy files.
  - `mmap()` allows file transfers without copying into user space.

mmap() Example: Fast File Copy

```
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * mmap.c - a program that uses mmap
 * to copy its source code to stdout
 */
```

```
int main() {
  struct stat stat;
  int i, fd, size;
  char *bufp;

  /* open the file & get its size*/
  fd = open("./mmap.c", O_RDONLY);
  fstat(fd, &stat);
  size = stat.st_size;
  /* map the file to a new VM area */
  bufp = mmap(0, size, PROT_READ,
    MAP_PRIVATE, fd, 0);

  /* write the VM area to stdout */
  write(1, bufp, size);
  exit(0);
}
```

Exec() Revisited



To run a new program p in the current process using exec( ):

- Free vm_area_struct's and page tables for old areas.
- Create new vm_area_struct's and page tables for new areas.
  - Stack, BSS, data, text, shared libs.
  - Text and data backed by ELF executable object file.
  - BSS and stack initialized to zero.
- Set PC to entry point in .text
  - Linux will fault in code, data pages as needed.

Fork() Revisited

To create a new process using `fork()`:

- Make copies of the old process's mm_struct, vm_area_struct's, and page tables.
  - At this point the two processes share all of their pages.
  - How to get separate spaces without copying all the virtual pages from one space to another?
    - "Copy on Write" (COW) technique.
- Copy-on-write
  - Mark PTE's of writeable areas as read-only
  - Writes by either process to these pages will cause page faults
  - Flag vm_area_struct's for these areas as private "copy-on-write"
    - Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

Net result:

- Copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

Memory System Summary

L1/L2 Memory Cache

- Purely a speed-up technique
- Behavior invisible to application programmer and (mostly) OS
- Implemented totally in hardware

Virtual Memory

- Supports many OS-related functions
  - Process creation, task switching, protection
- Software
  - Allocates/shares physical memory among processes
  - Maintains high-level tables tracking memory type, source, sharing
  - Handles exceptions, fills in hardware-defined mapping tables
- Hardware
  - Translates V➞P via mapping tables, enforcing permissions
  - Accelerates mapping via translation cache (TLB)

Further Reading

Intel TLBs

- Application Note: “TLBs, Paging-Structure Caches, and Their Invalidation”, April 2007