CIS 450 – Computer Architecture and Organization

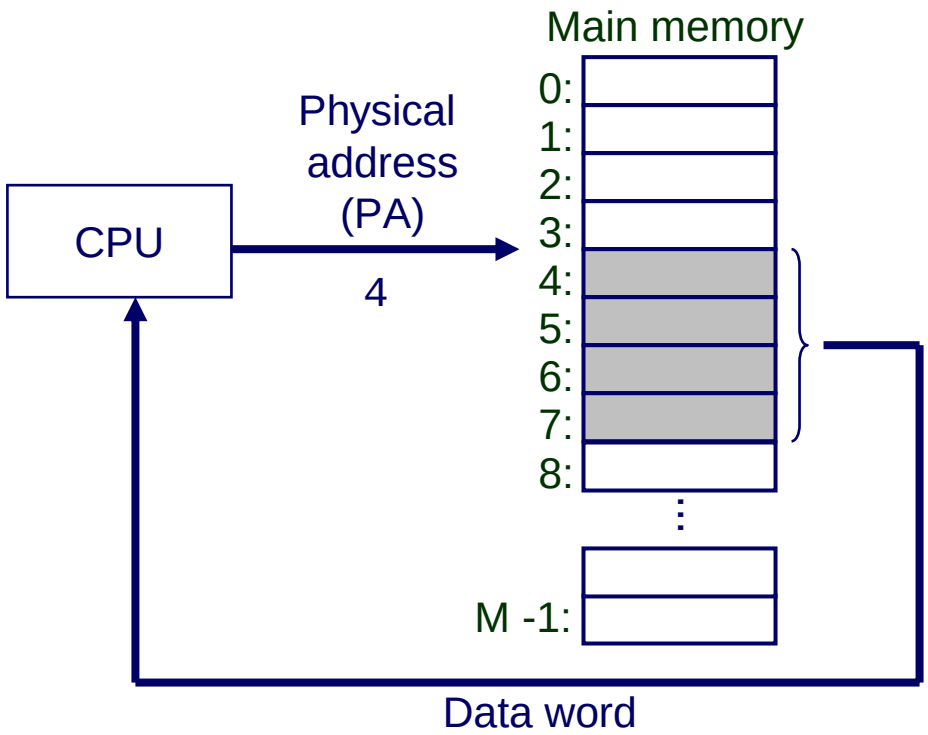
Lecture 23: Virtual Memory

Mitch Neilsen
(neilsen@ksu.edu)
219D Nichols Hall

Topics

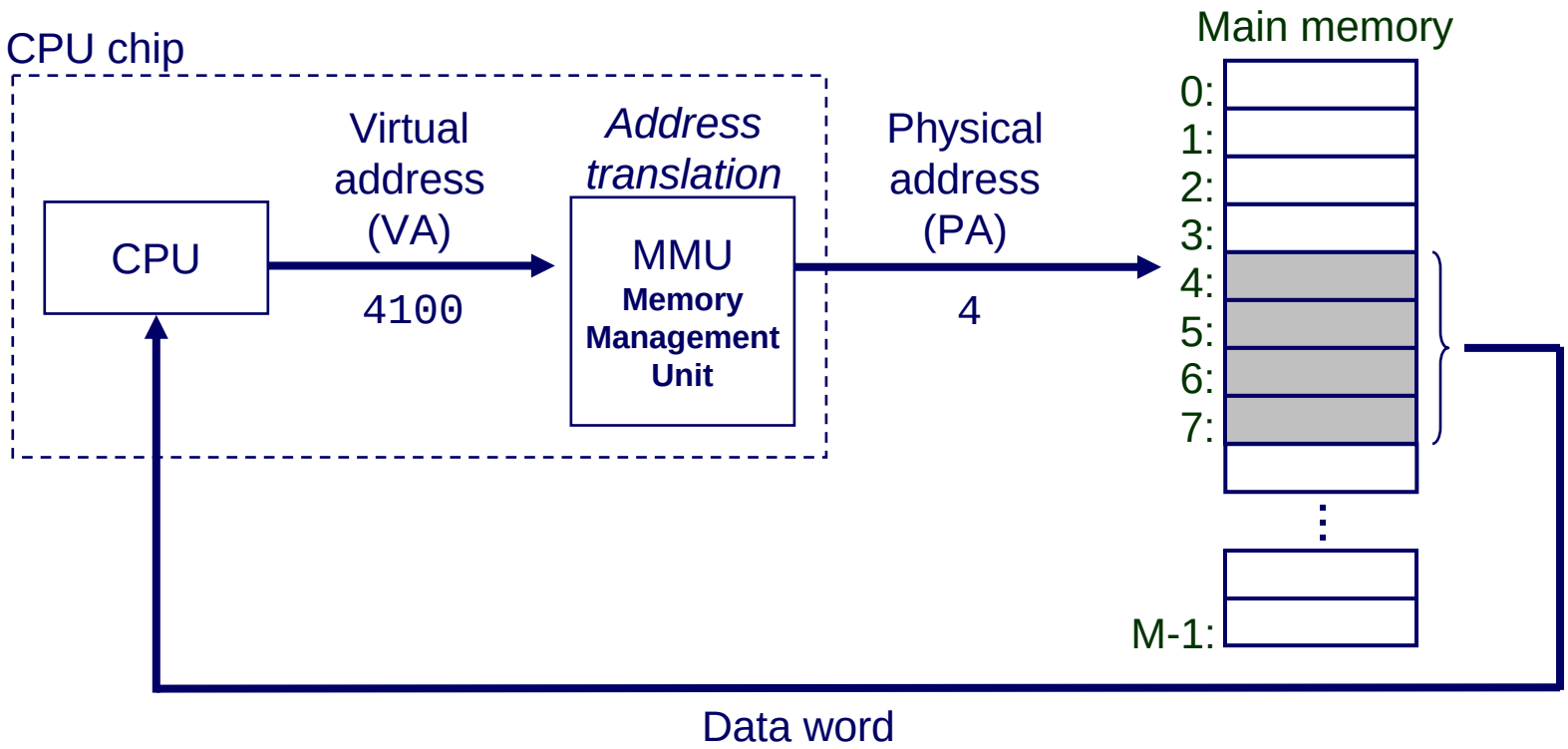
- **Virtual Memory**
  - **For Caching**
  - **For Memory Management**
- **Simple Virtual Memory System Example**
- **Intel P6 address translation**
- **x86-64 extensions**
- **Linux memory management**
- **Linux page fault handling**
- **Memory mapping**

A System Using Physical Addressing



Used by many digital signal processors and embedded microcontrollers in devices like phones and PDAs.

A System Using Virtual Addressing



One of the great ideas in computer science. Used by all modern desktop and laptop microprocessors.

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.

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.

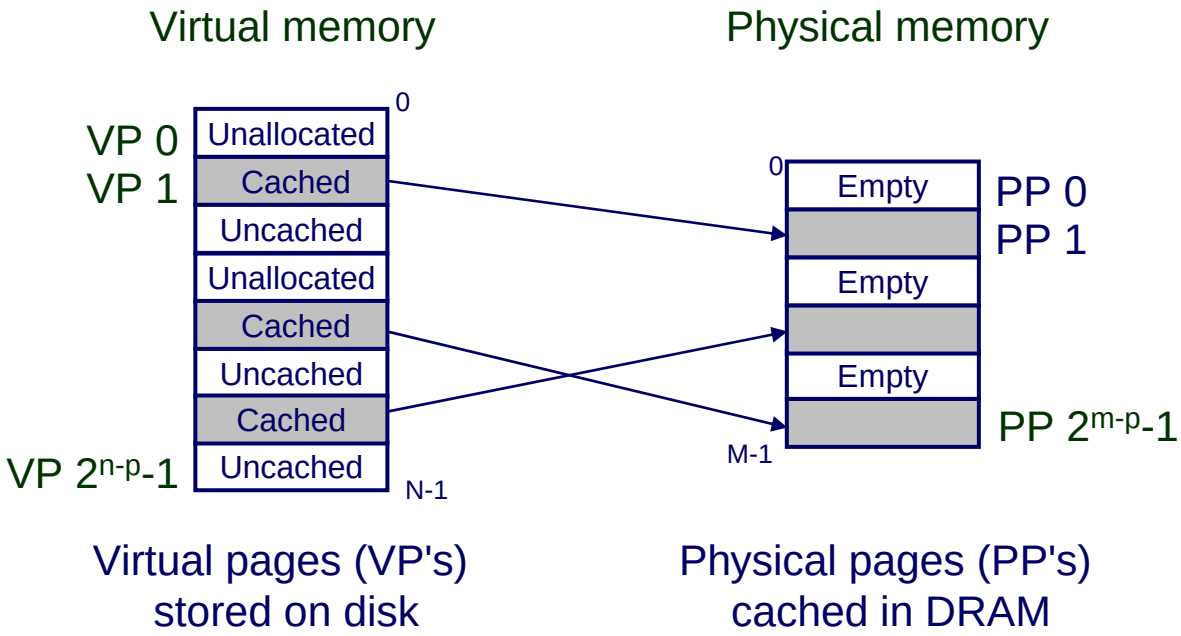
(3) VM protects address spaces

- 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.

(1) VM as a Tool for Caching

***Virtual memory* is an array of N contiguous bytes stored on disk.**

The contents of the array on disk are cached in *physical memory (DRAM cache)*



DRAM Cache Organization

DRAM cache organization driven by the enormous miss penalty

- **DRAM is about 10x slower than SRAM**
- **Disk is about 100,000x slower than a DRAM**

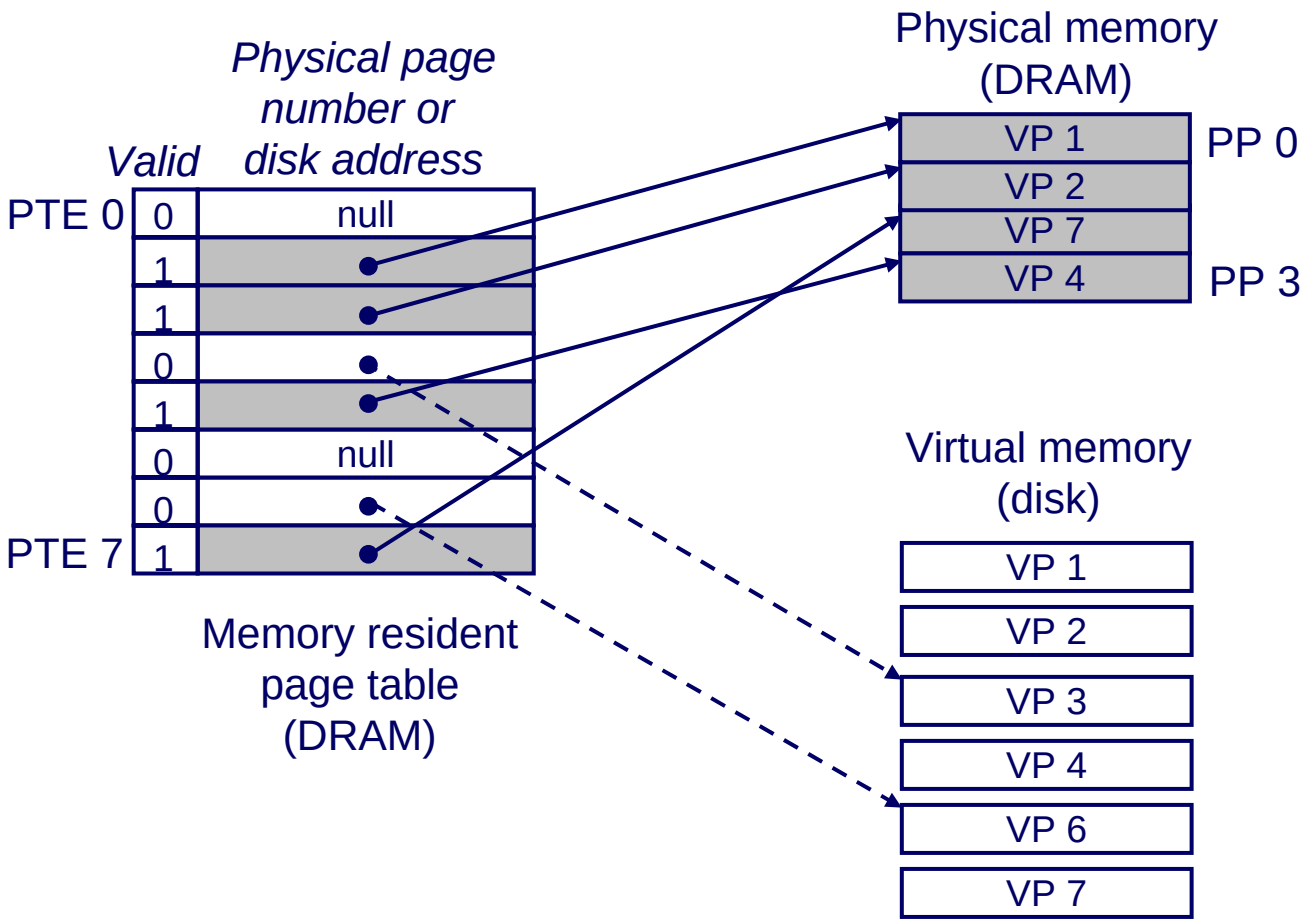
DRAM cache properties

- **Large page (block) size (typically 4-8 KB)**
- **Fully associative**
  - **Any virtual page can be placed in any physical page**
- **Highly sophisticated replacement algorithms**
- **Write-back rather than write-through**

Page Tables

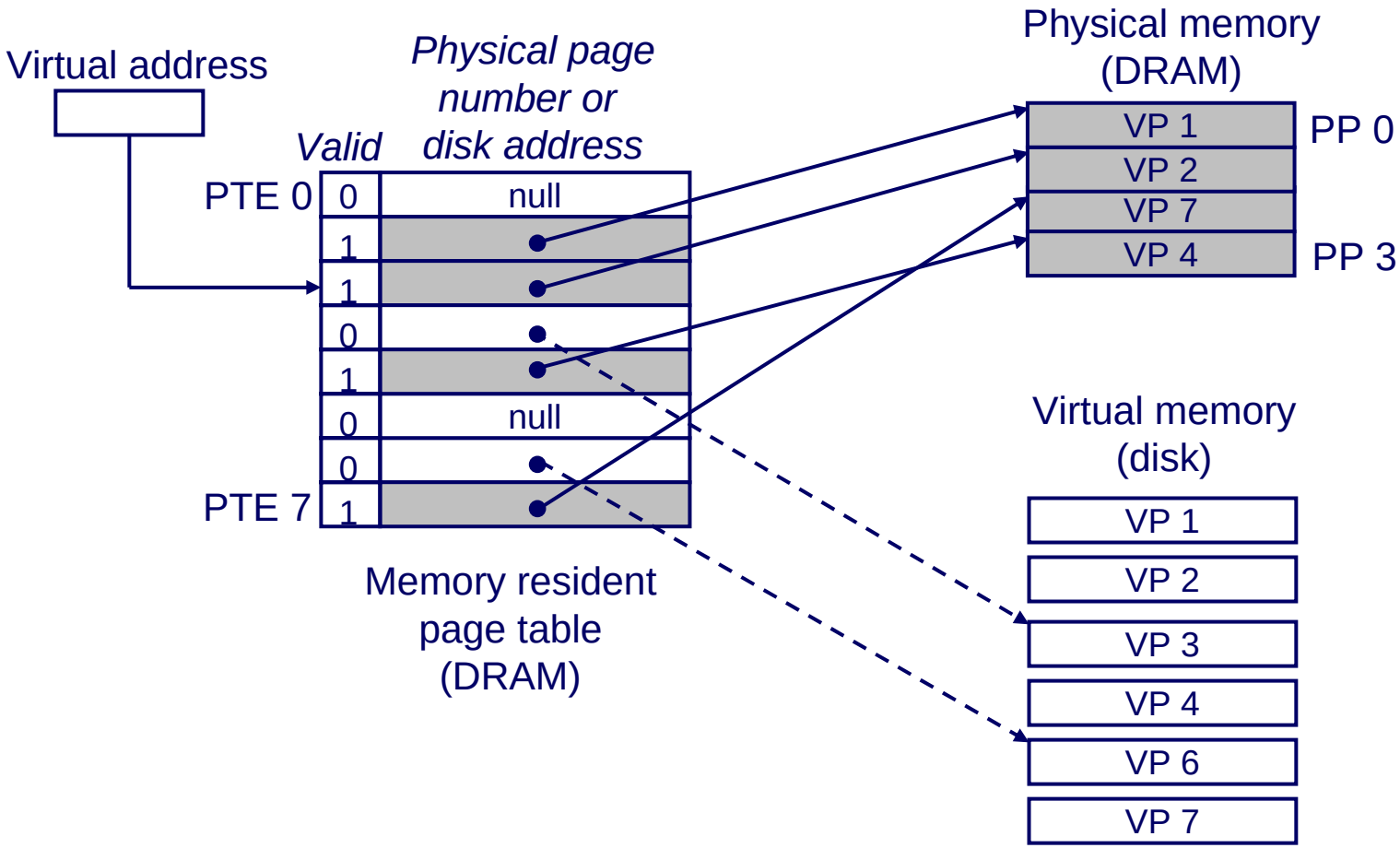
A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.

- **Kernel data structure in DRAM**



Page Hits

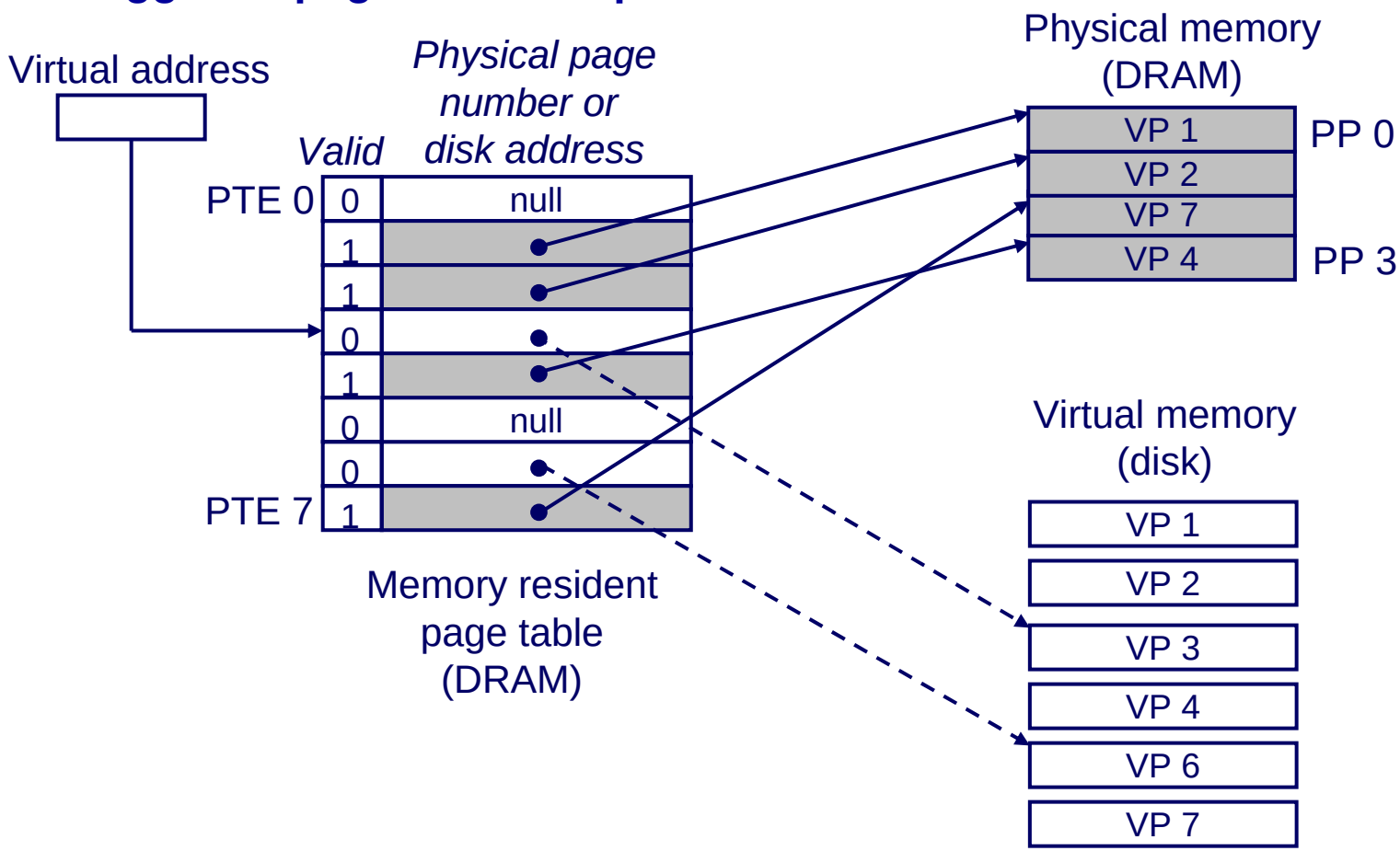
A *page hit* is a reference to a VM word that is in physical (main) memory.



Page Faults

A *page fault* is caused by a reference to a VM word that is not in physical (main) memory.

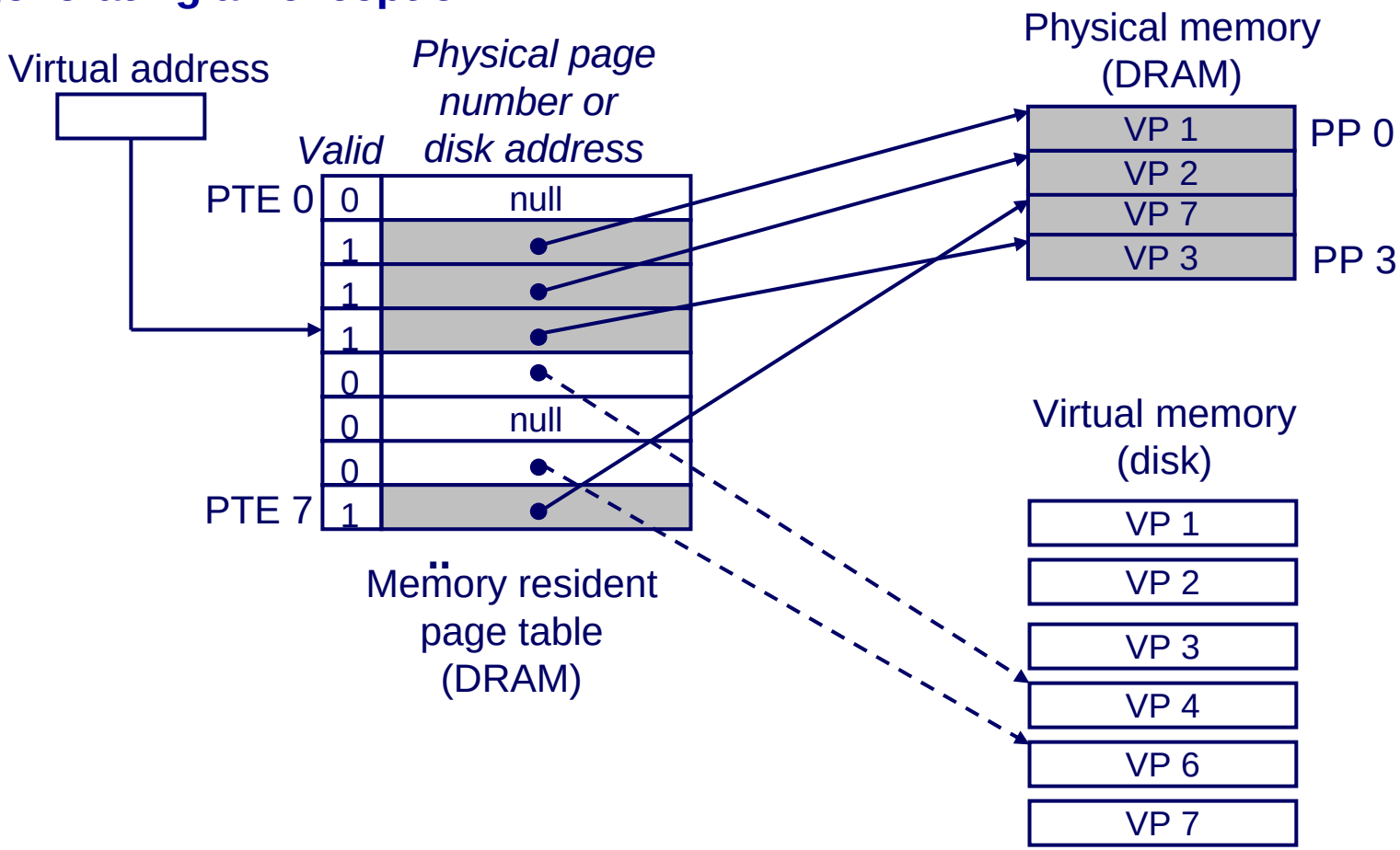
- **Example: A instruction references a word contained in VP 3, a miss that triggers a page fault exception**



Page Faults (cont)

The kernel's page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk (*demand paging*)

- **When the offending instruction restarts, it executes normally, without generating an exception**



Servicing a Page Fault

(1) Processor signals controller

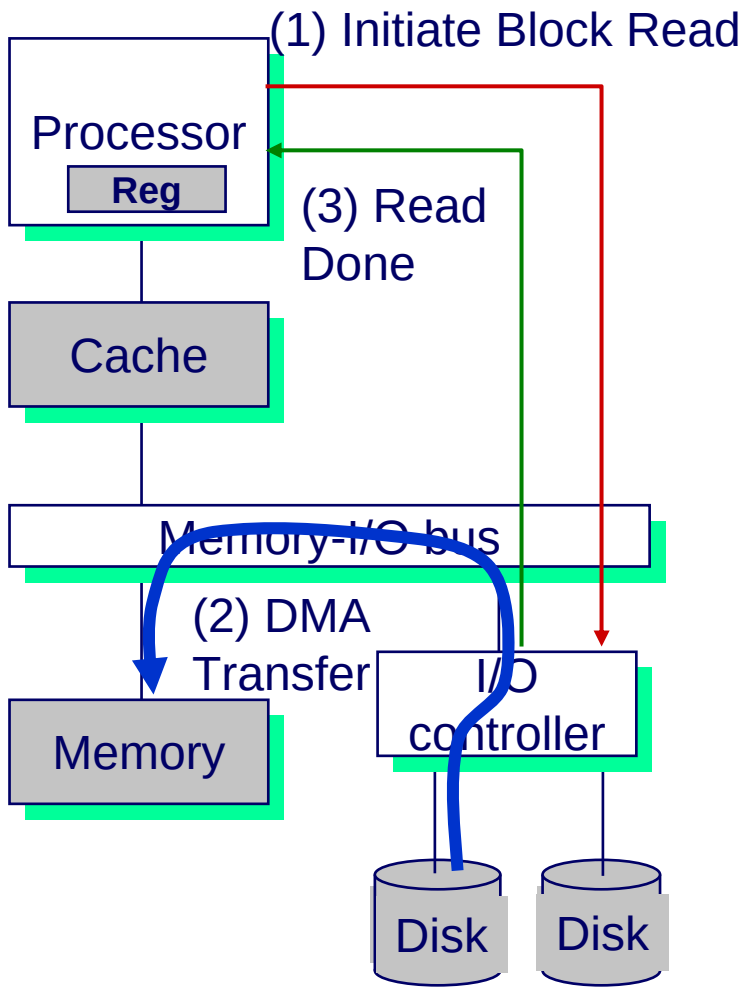
- **Read block of length P starting at disk address X and store starting at memory address Y**

(2) Read occurs

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

(3) Controller signals completion

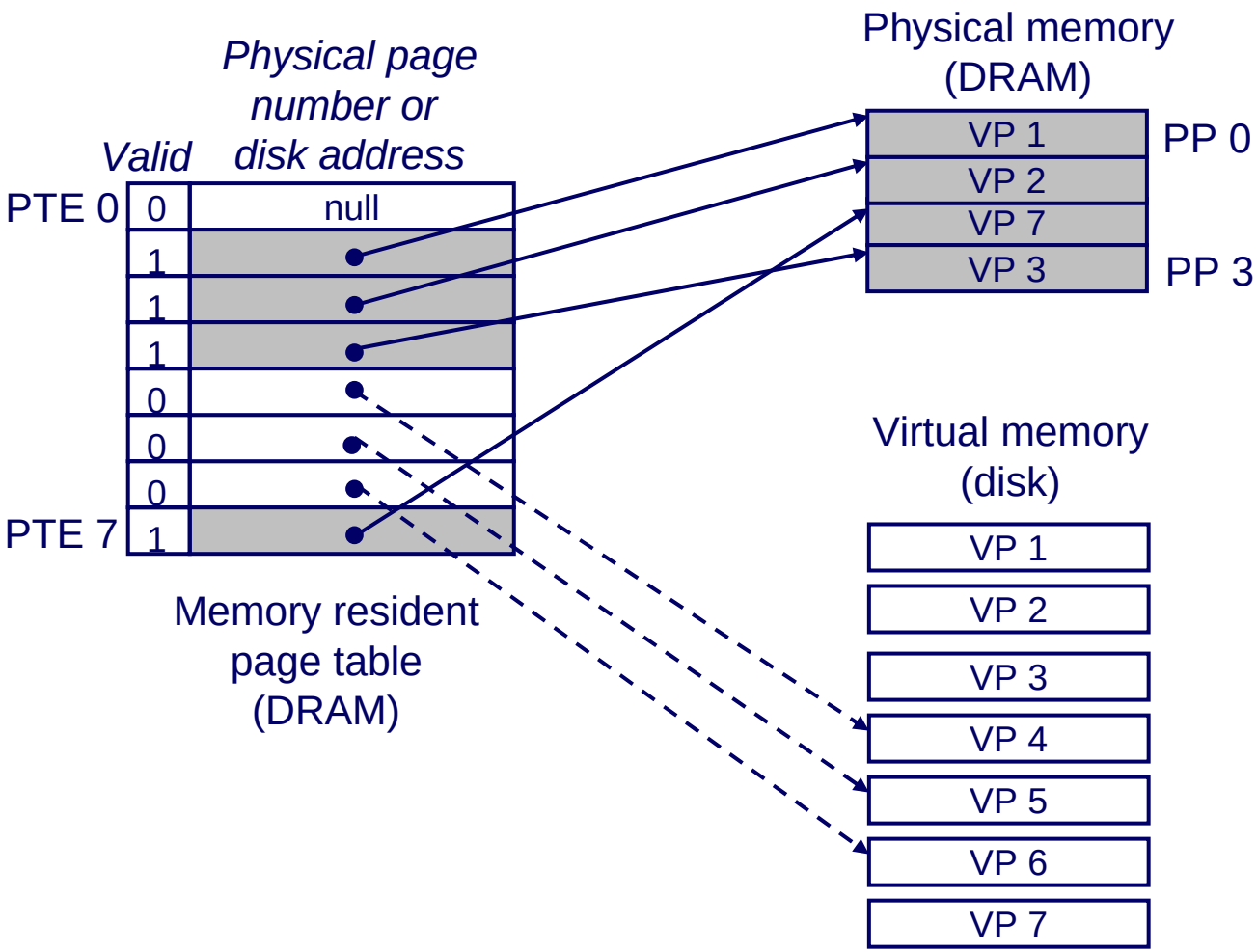
- **Interrupt processor**
- **OS resumes suspended process**



Allocating Virtual Pages

Example: Allocating new virtual page VP5

- **Kernel allocates VP 5 on disk and points PTE 5 to this new location.**



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.**

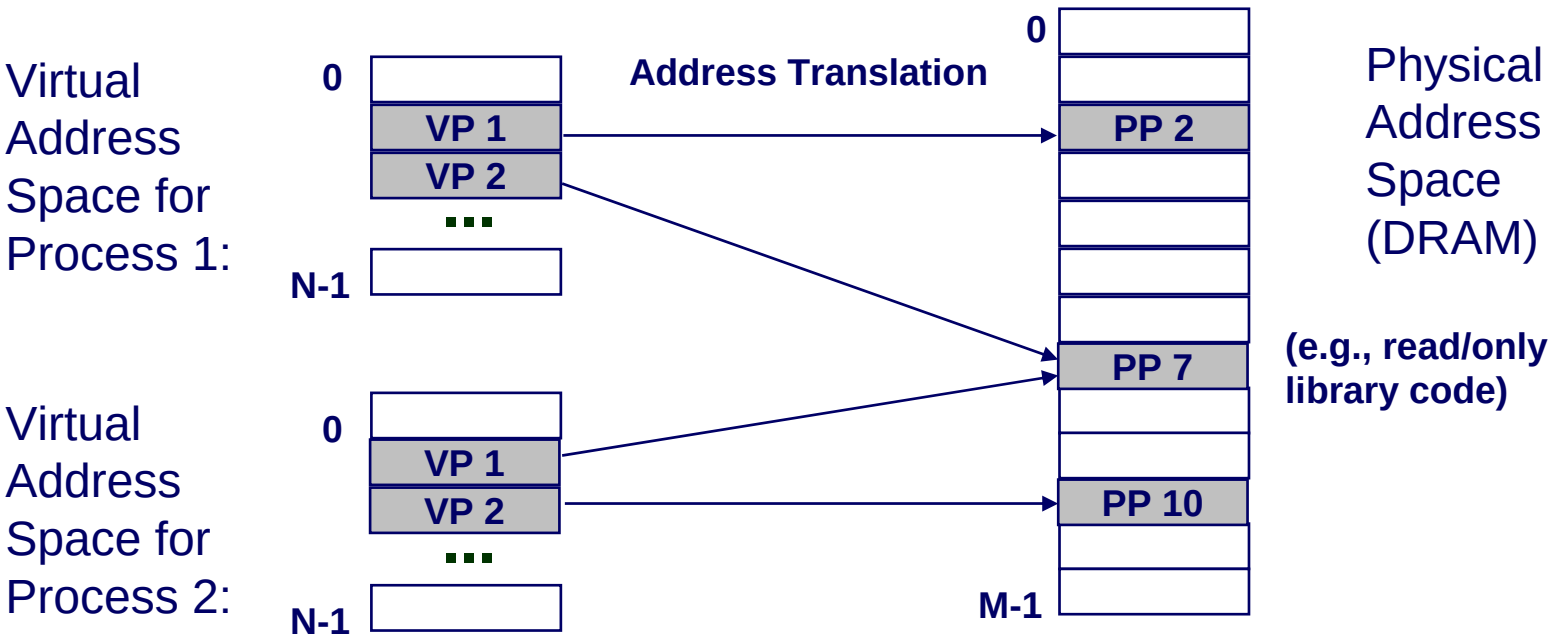
If working set size > main memory size

- ***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.**



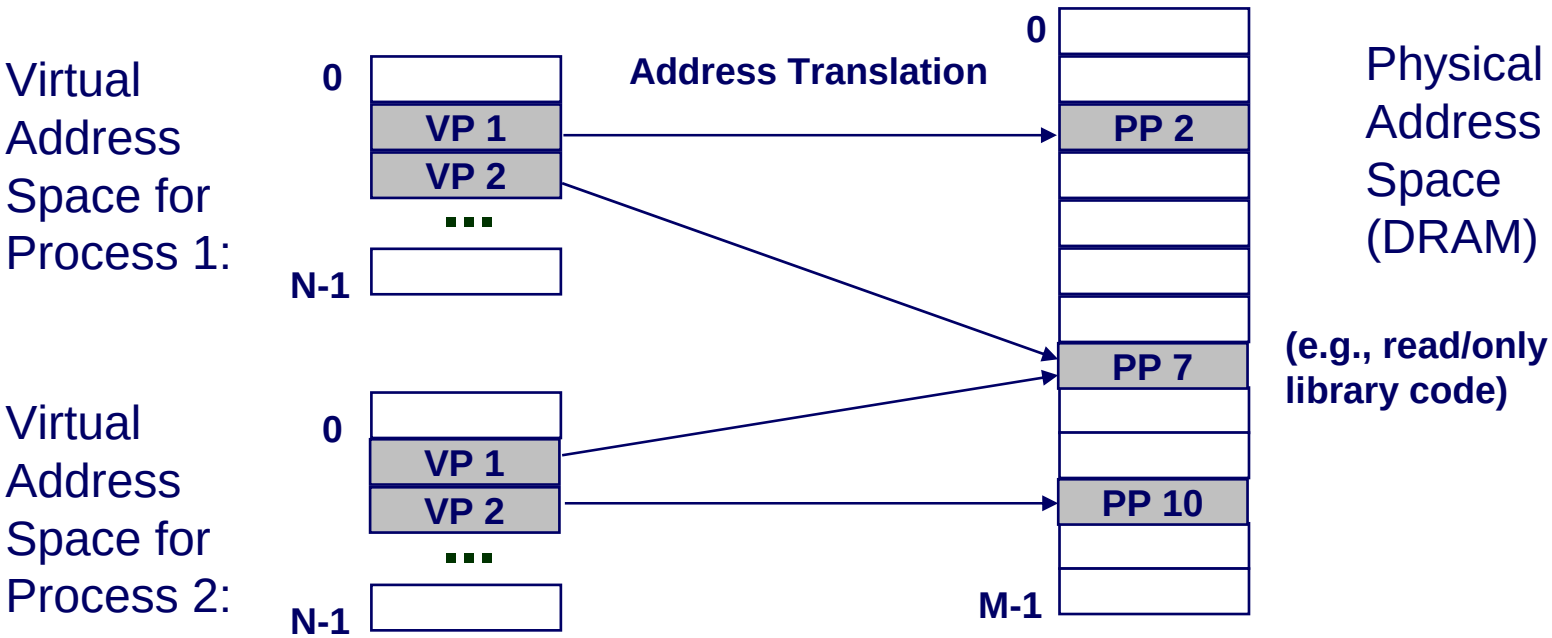
Simplifying Sharing and Allocation

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



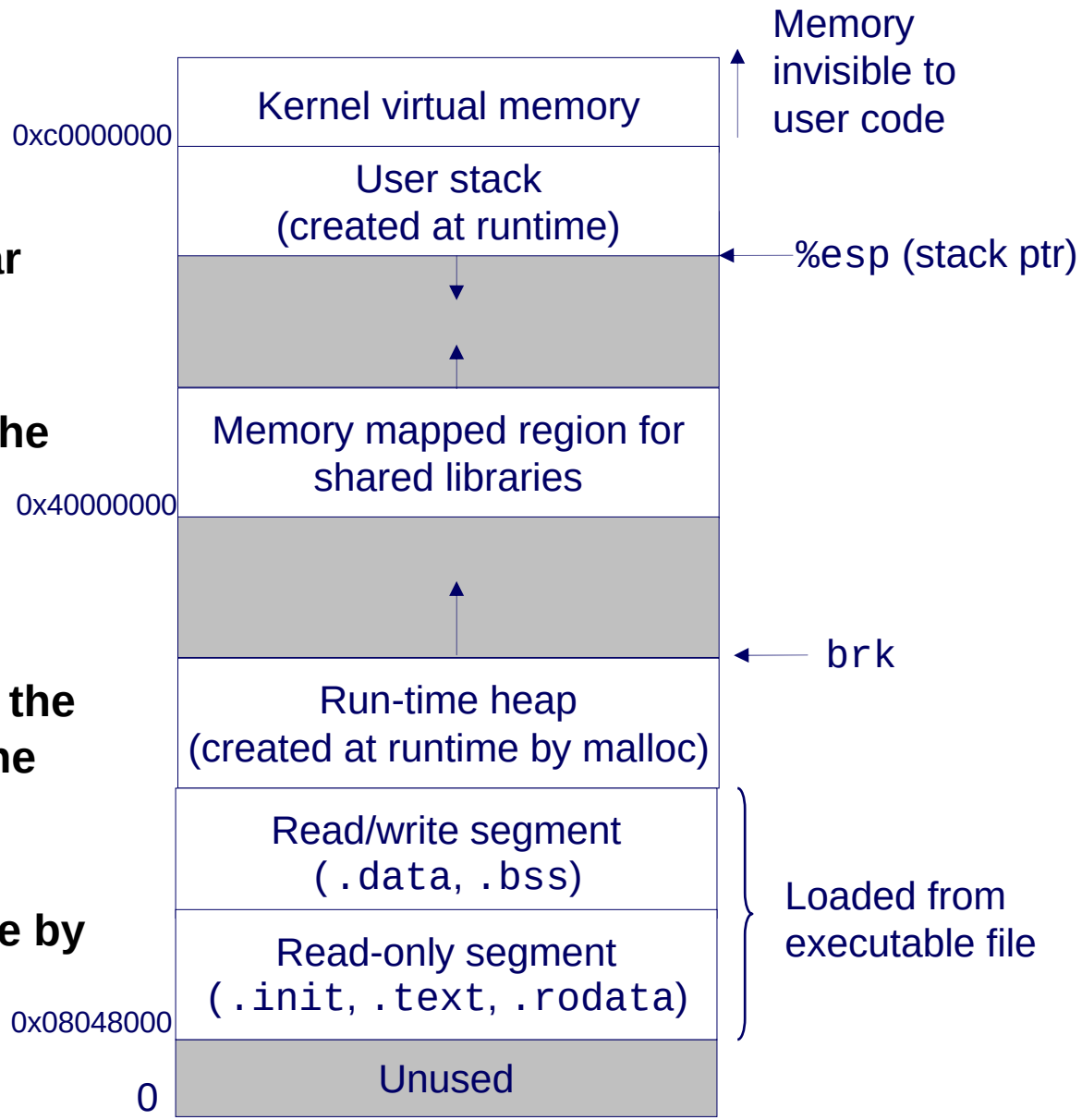
Simplifying Linking and Loading

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.**

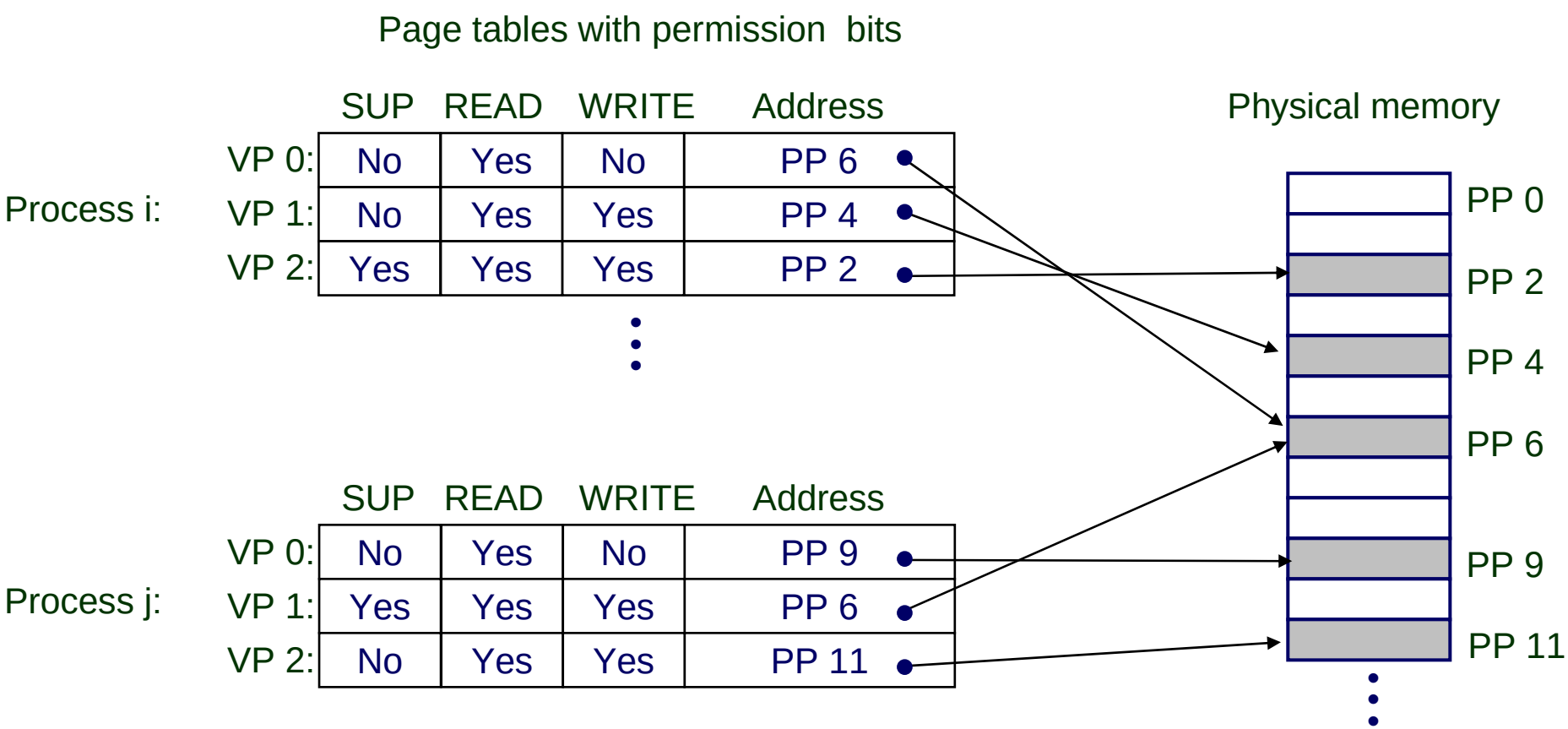


(3)VM as a Tool for Memory Protection

Extend PTEs with permission bits.

Page fault handler checks these before remapping.

- **If violated, send process SIGSEGV (segmentation fault)**



VM Address Translation

Virtual Address Space

- V = {0, 1, …, N–1}

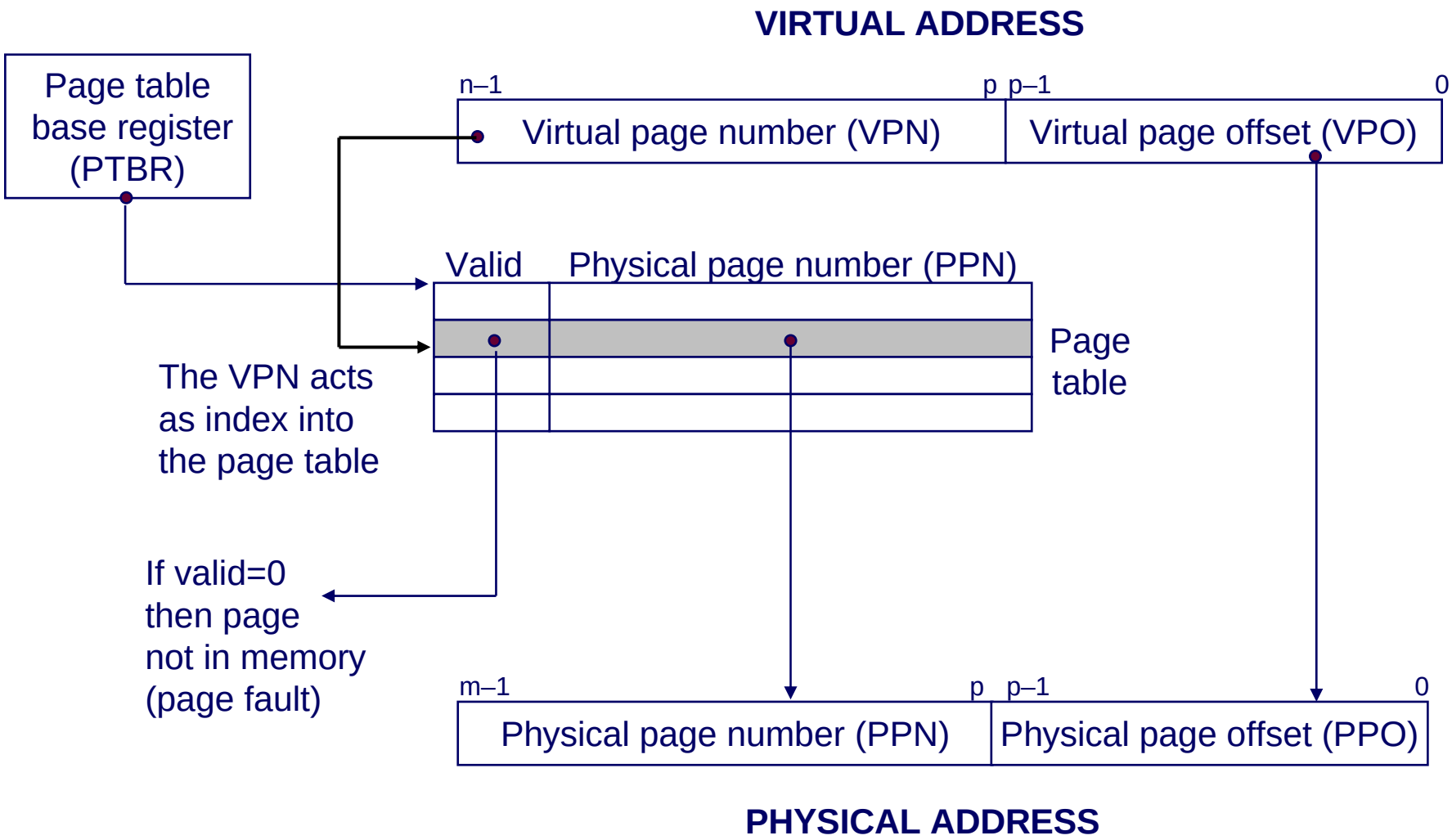
Physical Address Space

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

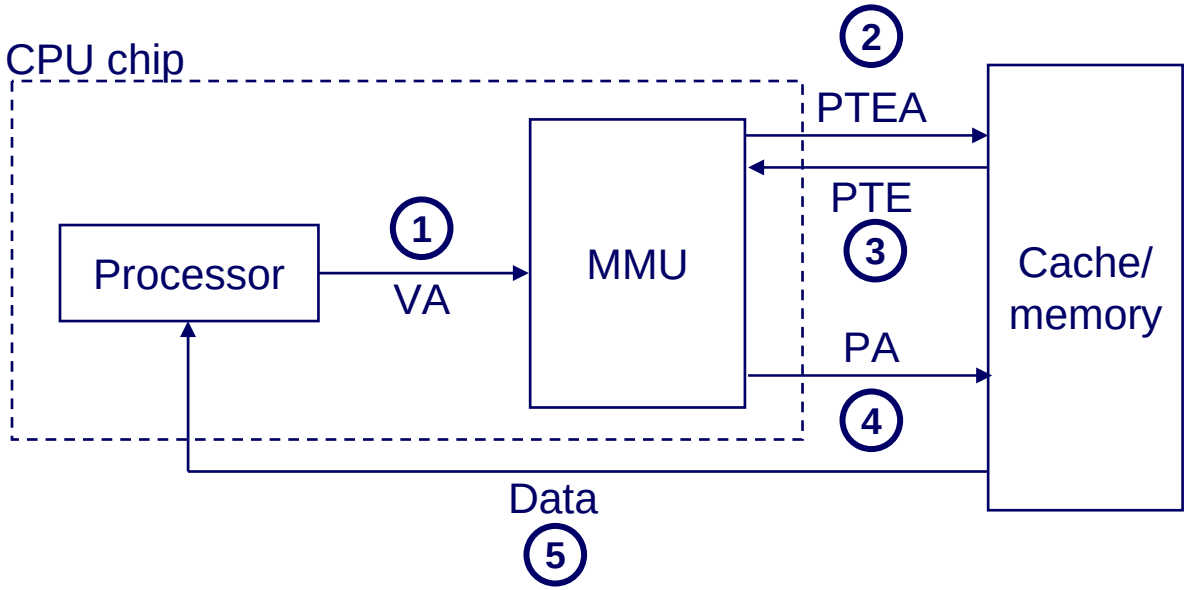
Address Translation

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

Address Translation with a Page Table



Address Translation: Page Hit



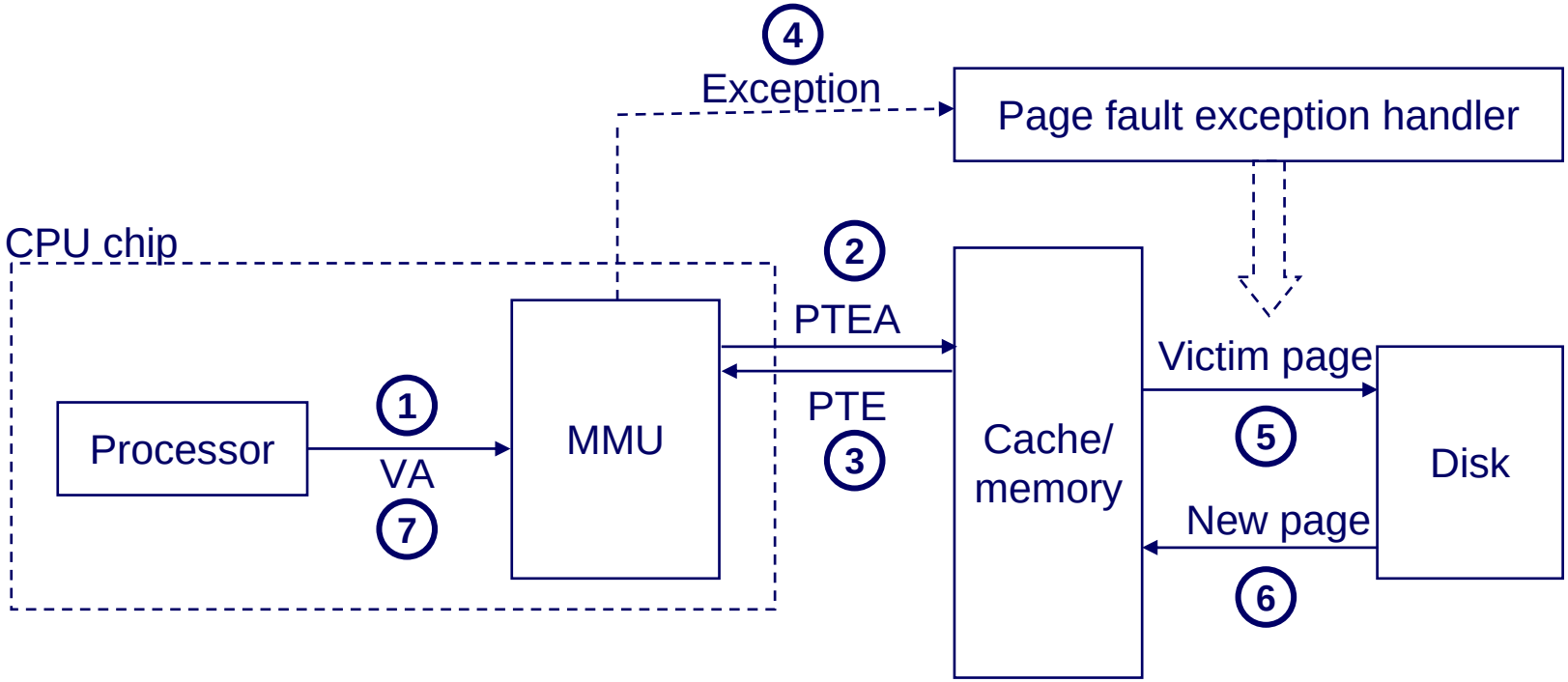
1) Processor sends virtual address to MMU

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

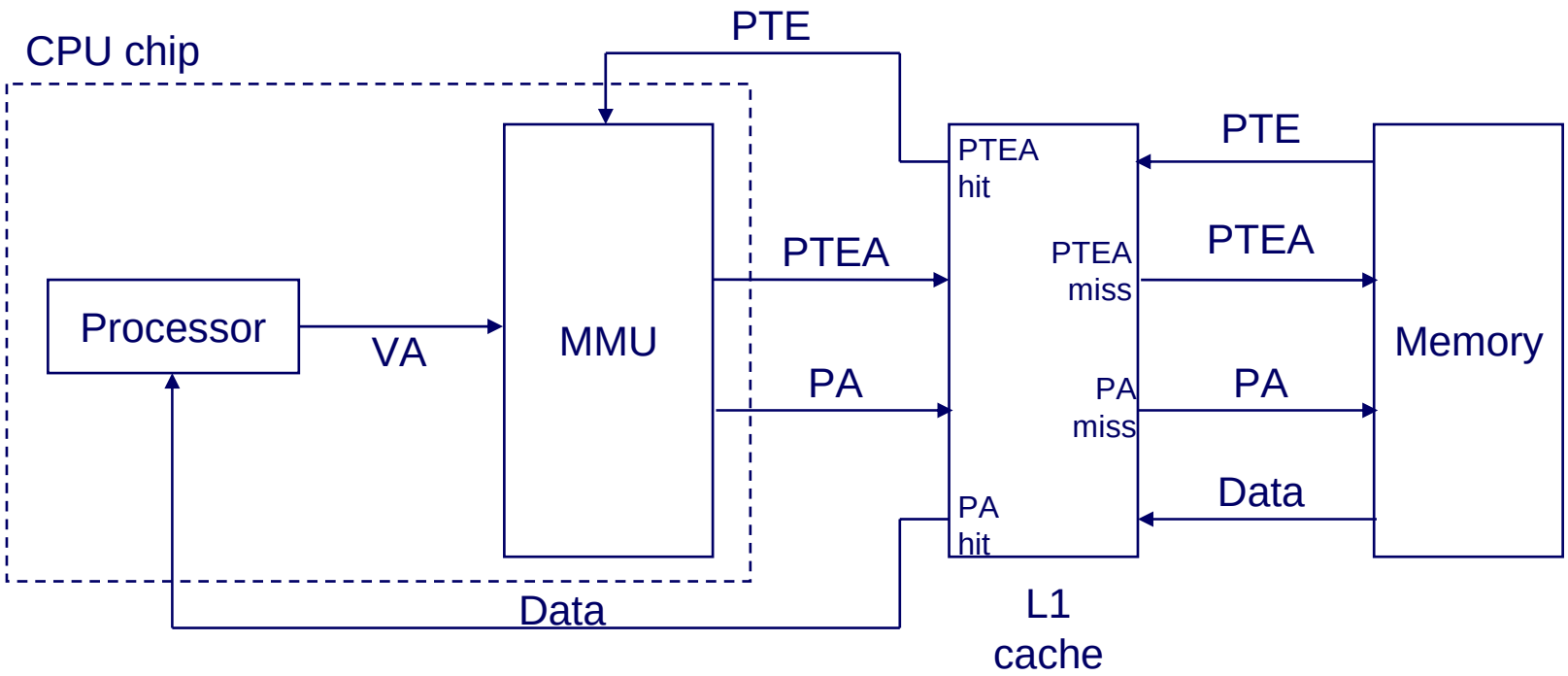
4) Valid bit is zero, so MMU triggers page fault exception

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.

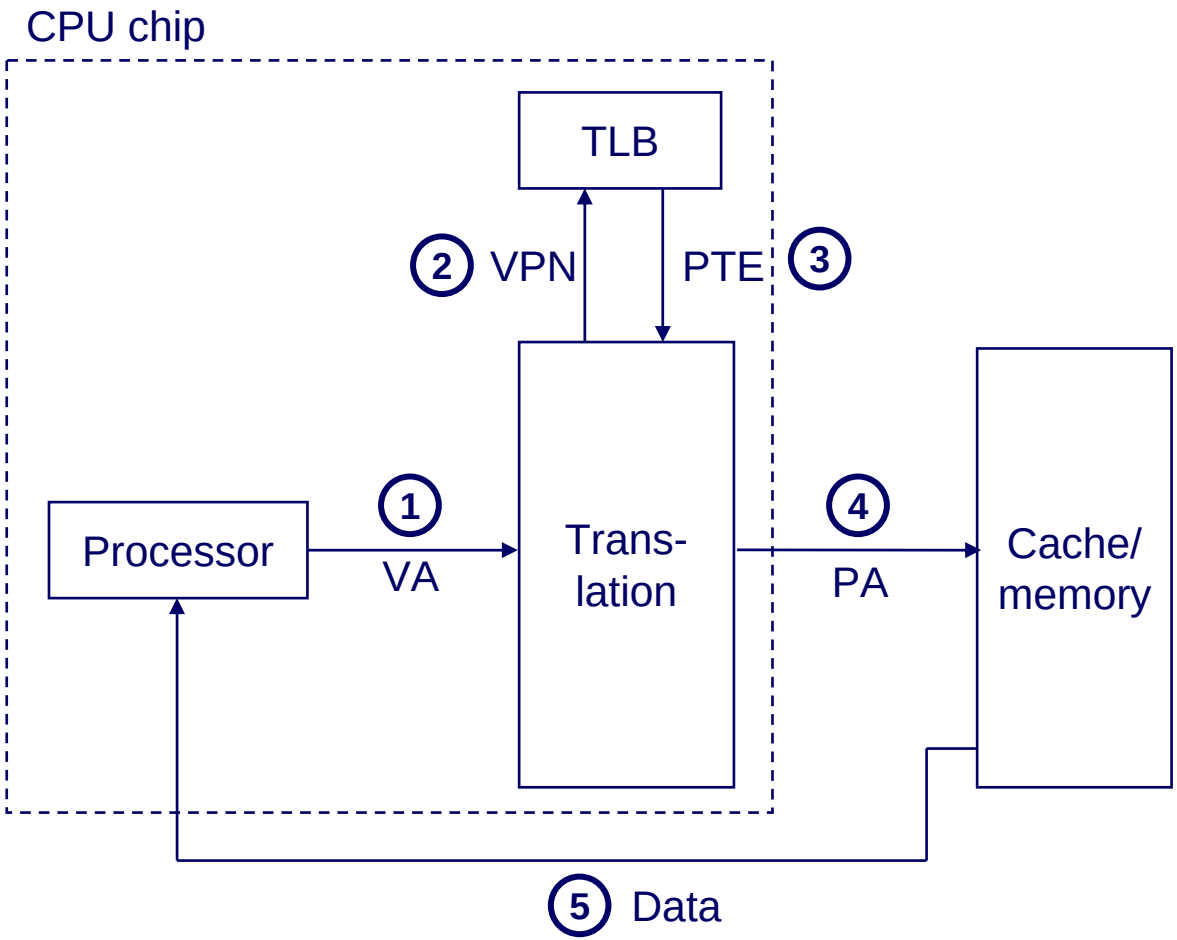
- **Translation Lookaside Buffer (TLB)**

Speeding up Translation with a TLB

***Translation Lookaside Buffer* (TLB)**

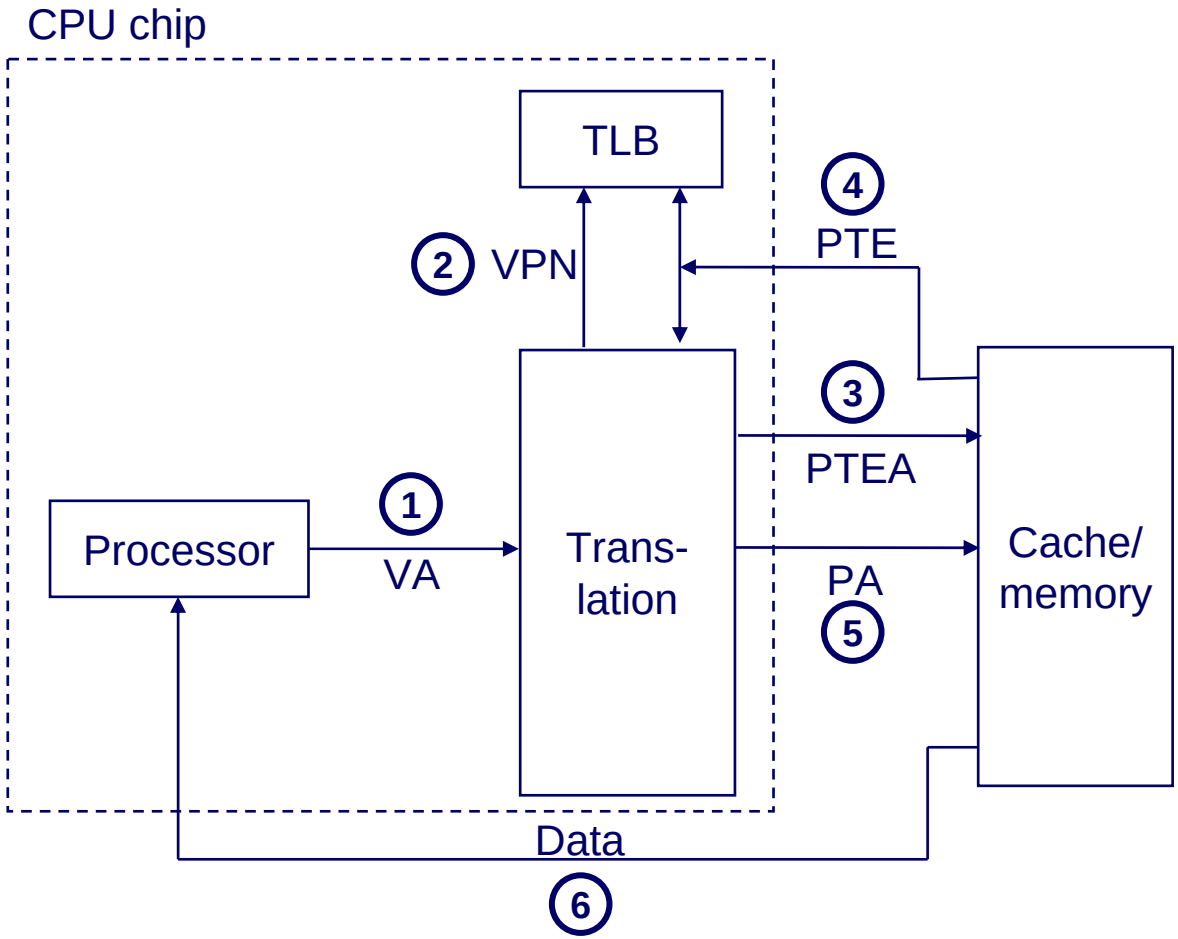
- **Small hardware cache in MMU**
- **Maps virtual page numbers to physical page numbers**
- **Contains complete page table entries for small number of pages**

TLB Hit



A TLB hit eliminates a memory access.

TLB Miss



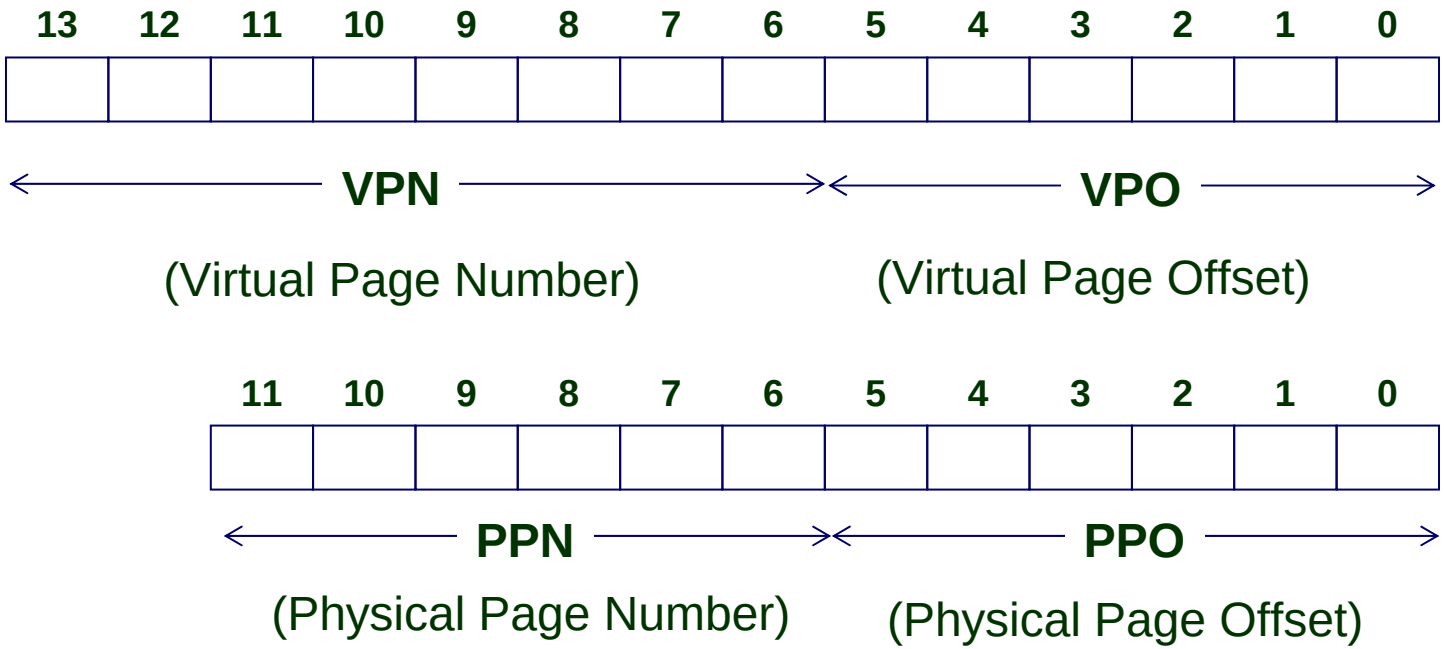
A TLB miss incurs an additional memory access (the PTE).

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

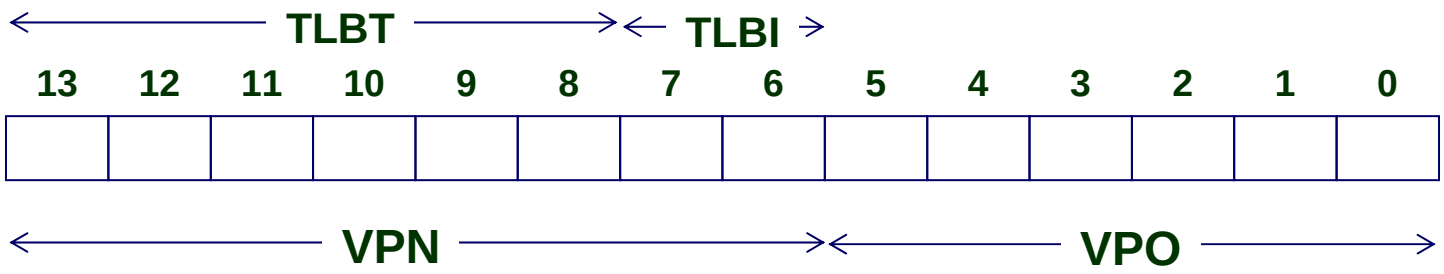
- **Only show first 16 entries (out of 256)**

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**

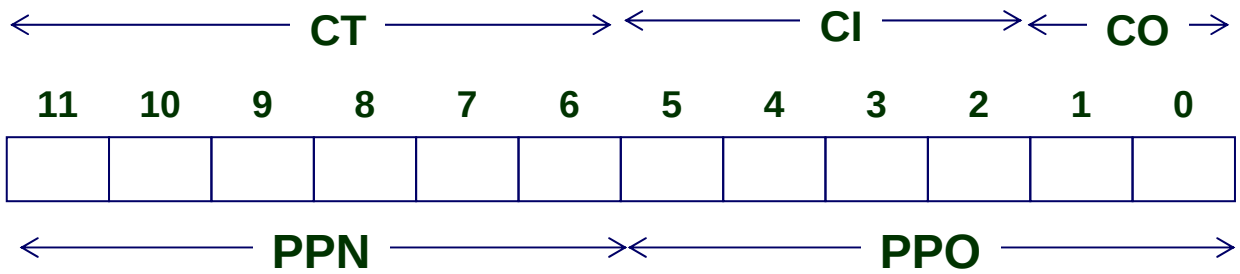


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System Cache

Cache

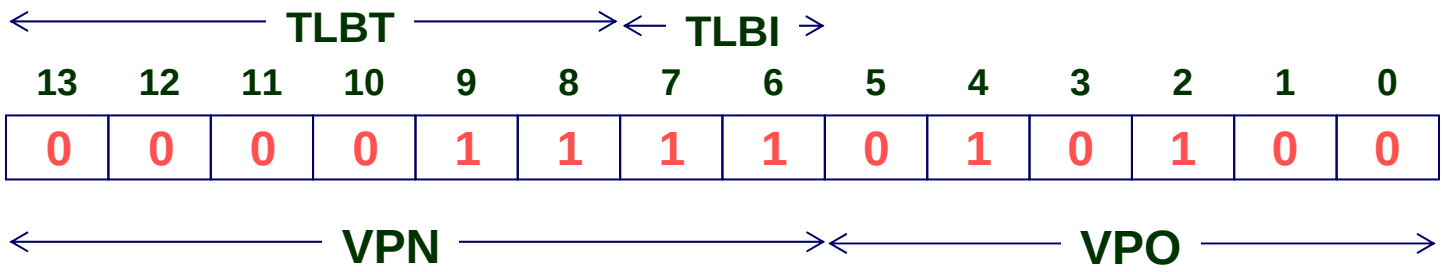
- 16 lines
- 4-byte line size
- Direct mapped



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	–	–	–	–

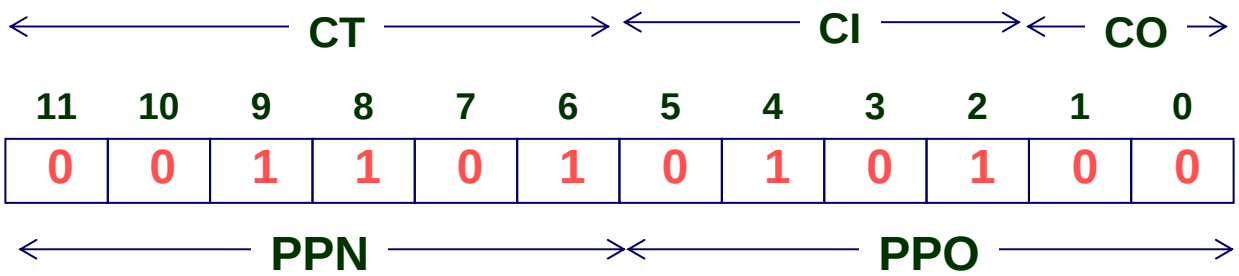
Address Translation Example #1

Virtual Address `0x03D4`



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

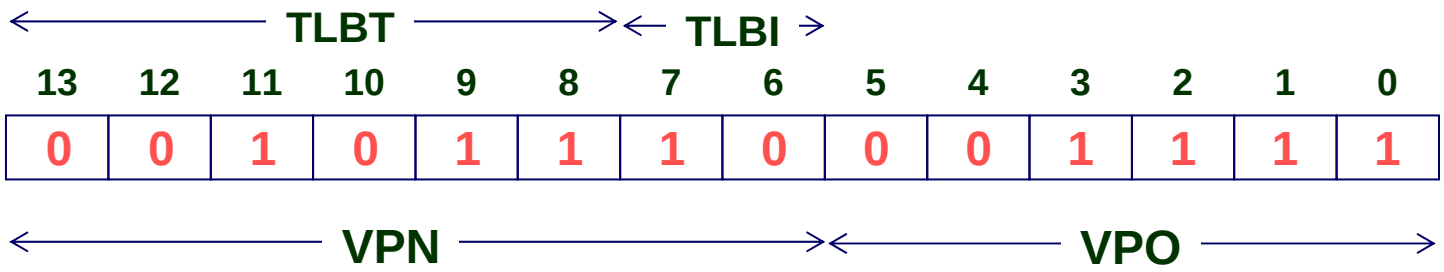
Physical Address



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

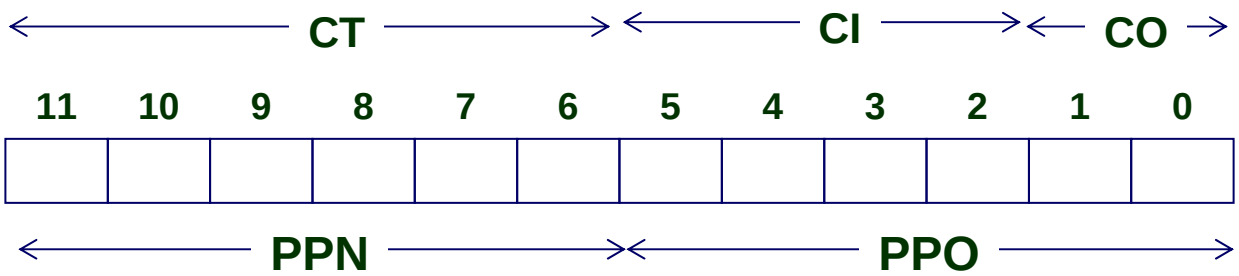
Address Translation Example #2

Virtual Address `0x0B8F`



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

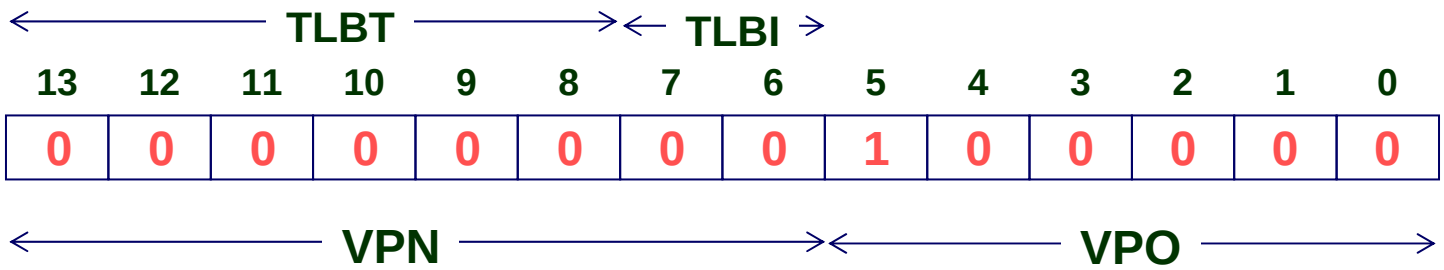
Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____

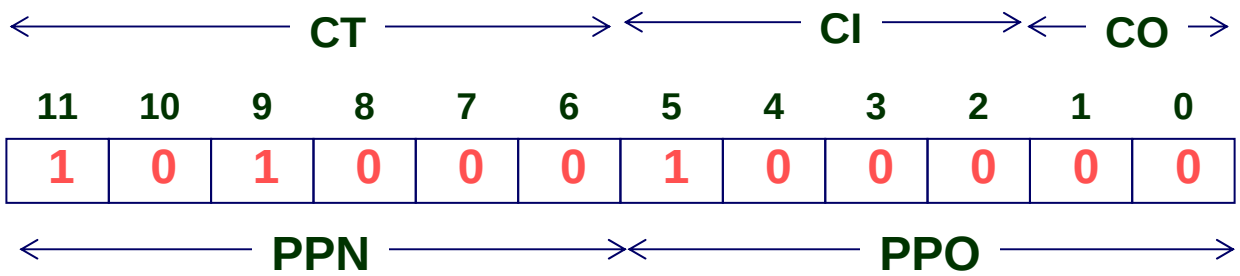
Address Translation Example #3

Virtual Address `0x0020`



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

Physical Address



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

Multi-Level Page Tables

Given:

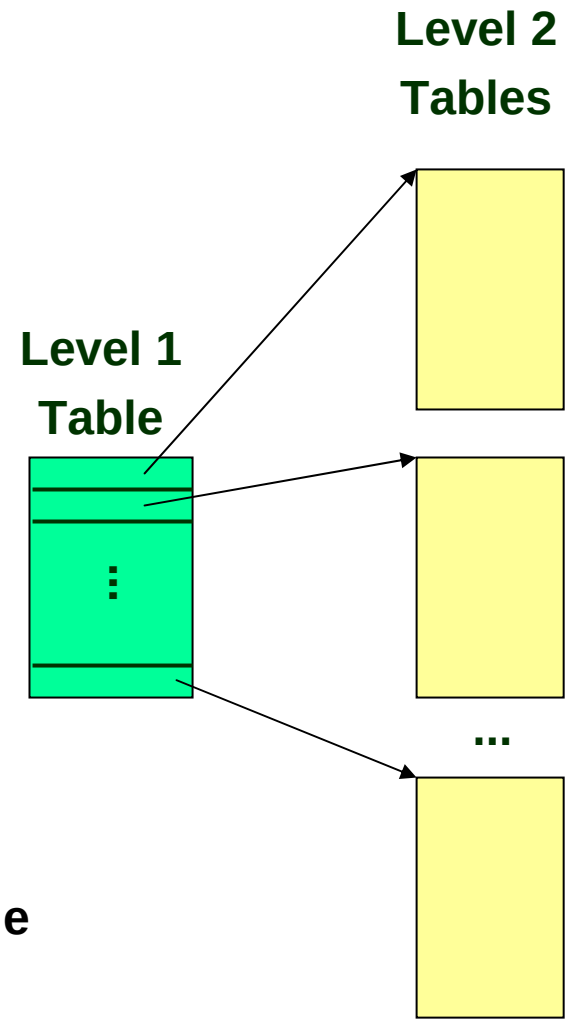
- 4KB (2^{12}) page size
- 48-bit address space
- 4-byte PTE

Problem:

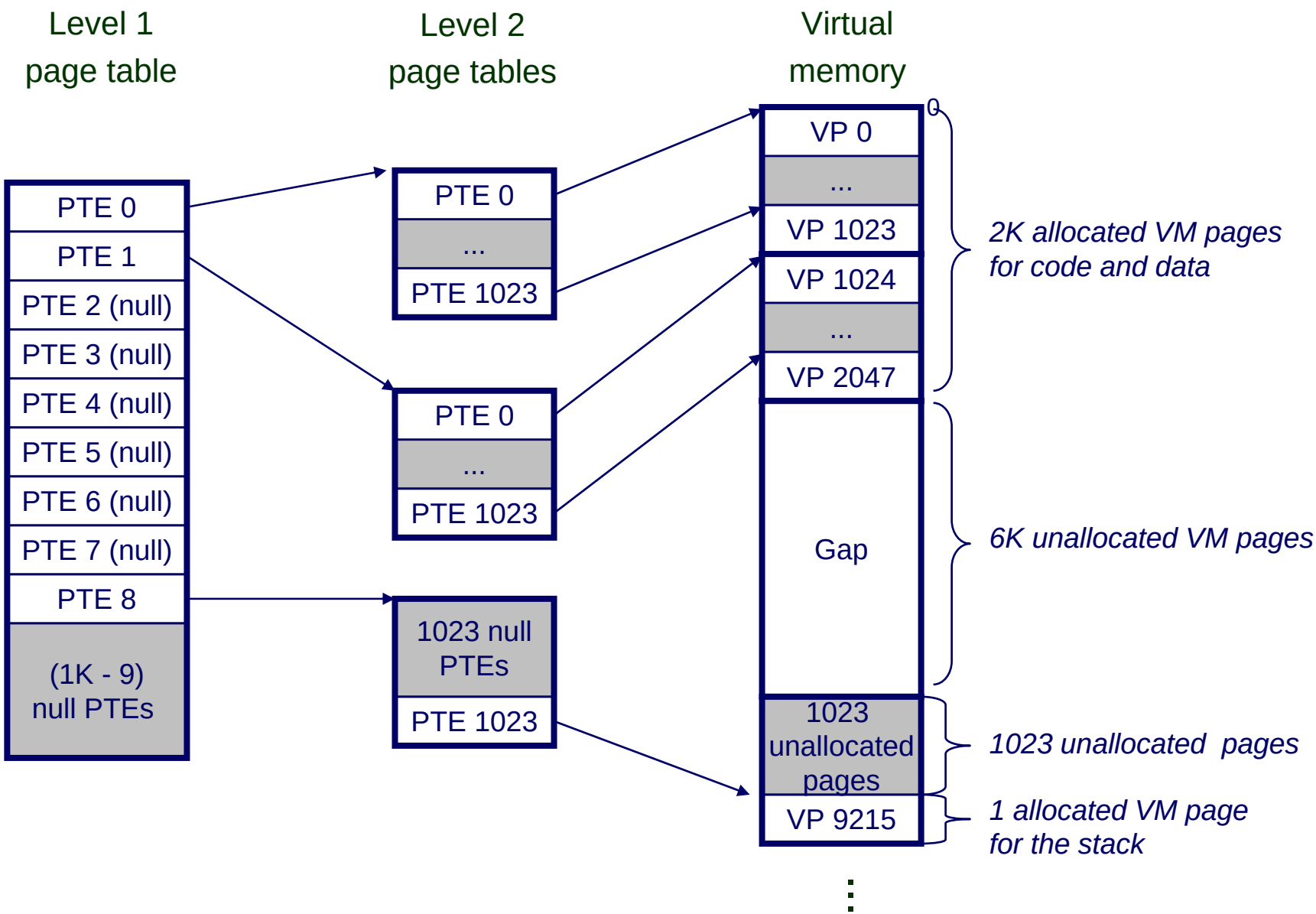
- Would need a 256 GB page table!
  - $2^{48} * 2^{-12} * 2^{2} = 2^{38}$ bytes

Common solution

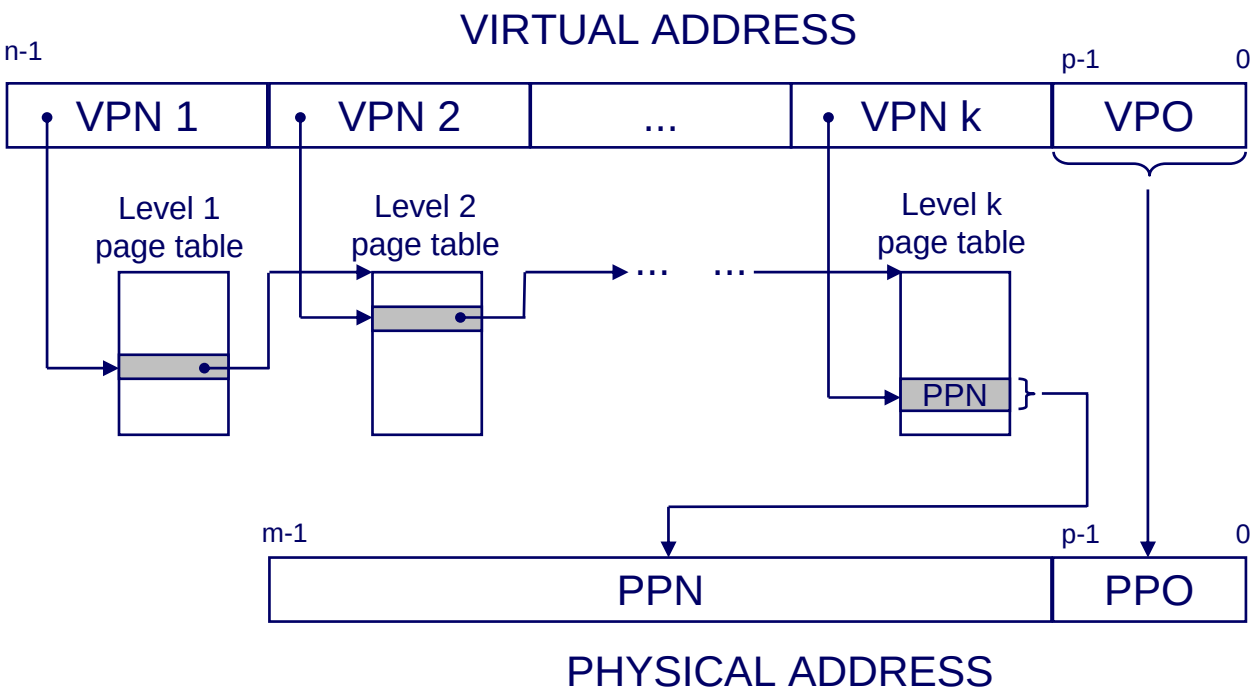
- Multi-level page tables
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (memory resident)
  - Level 2 table: Each PTE points to a page (paged in and out like other data)



A Two-Level Page Table Hierarchy



Translating with a k-level Page Table



Intel P6

Internal designation for successor to Itanium

- Itanium had internal designation P5

Fundamentally different from Pentium

- Out-of-order, superscalar operation

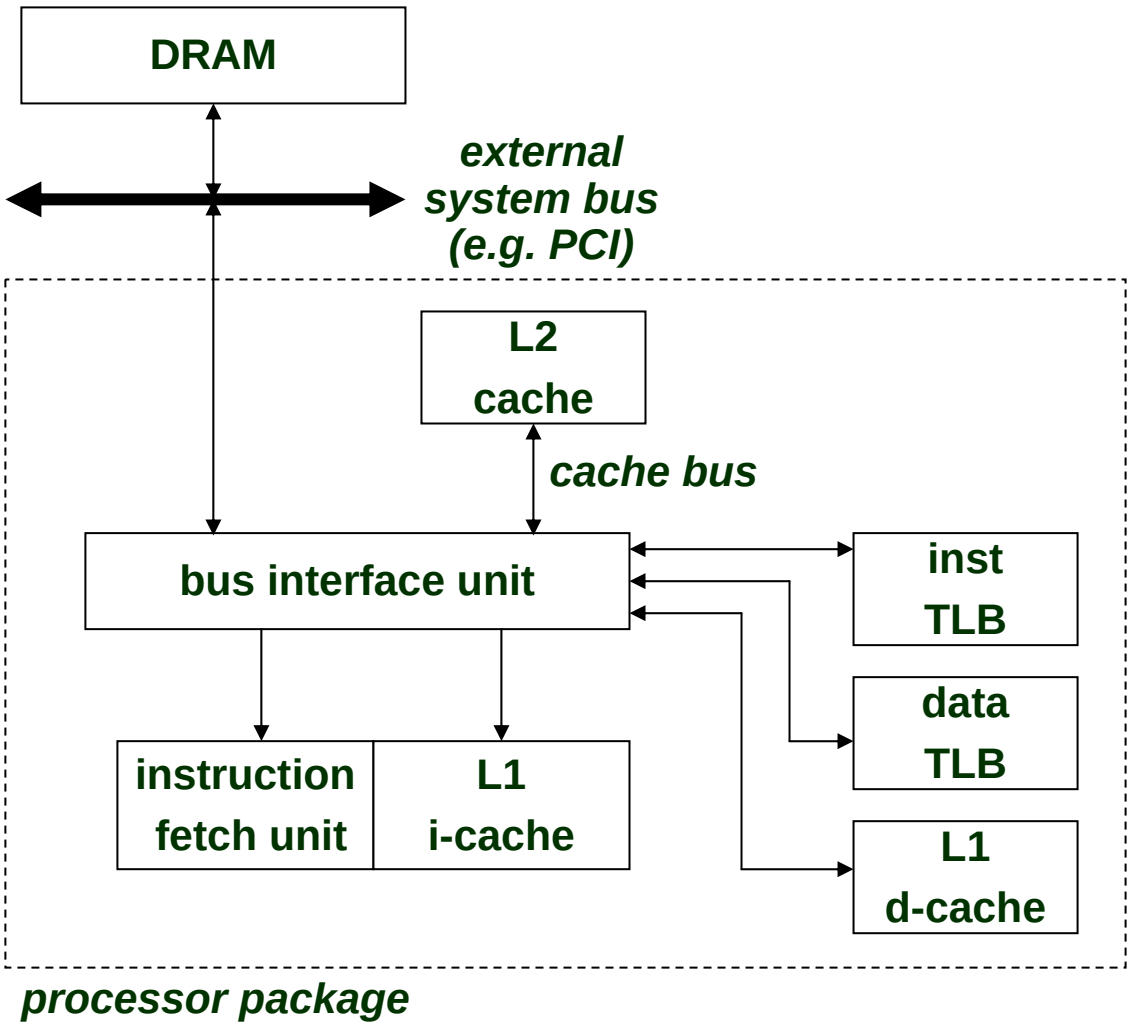
Resulting processors

- Pentium Pro (1996)
- Pentium II (1997)
  - L2 cache on same chip
- Pentium III (1999)

Pentium 4

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

P6 Memory System



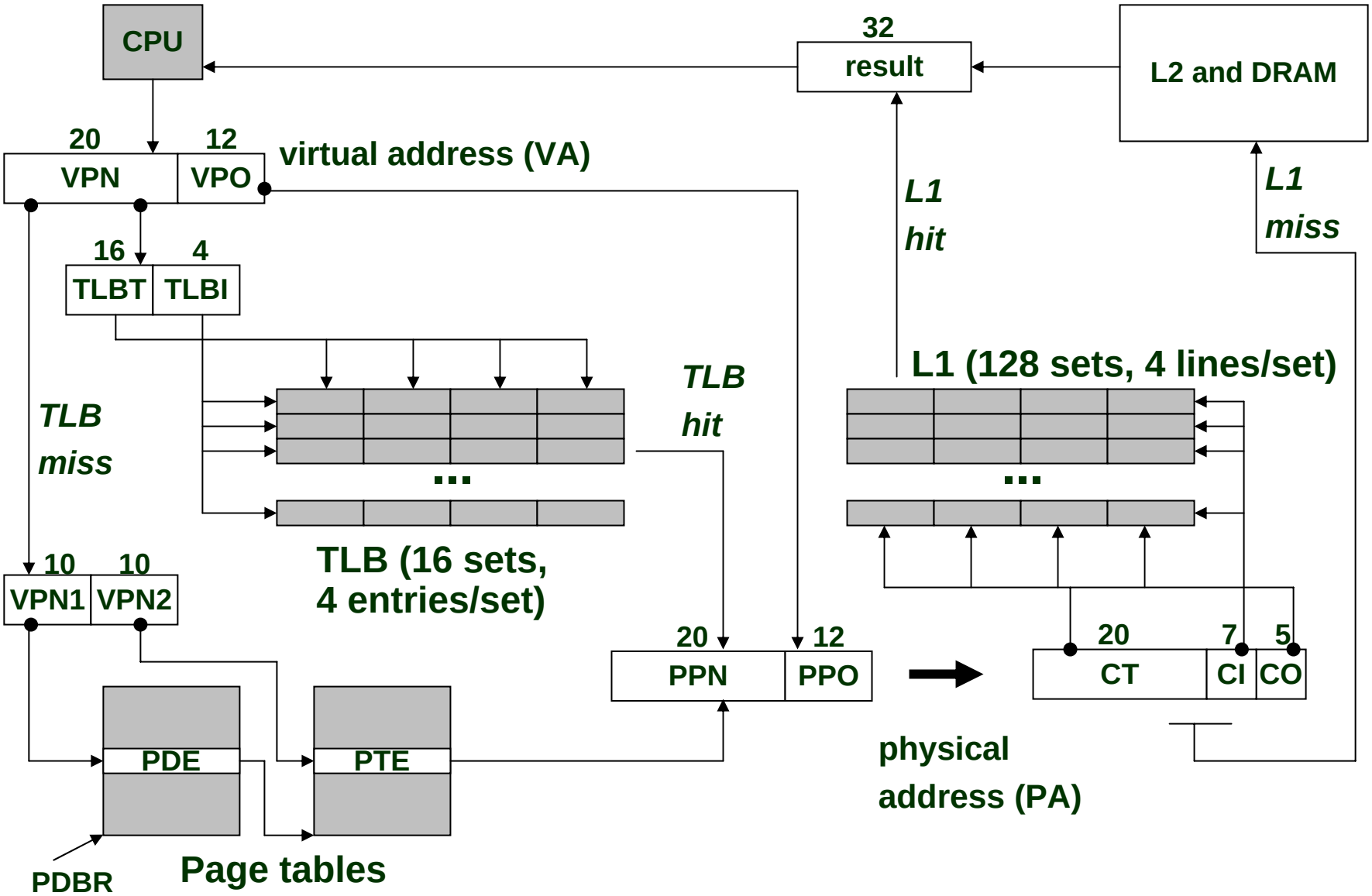
- 32 bit address space
- 4 KB page size
- L1, L2, and TLBs
  - 4-way set associative
- Inst TLB
  - 32 entries
  - 8 sets
- Data TLB
  - 64 entries
  - 16 sets
- L1 i-cache and d-cache
  - 16 KB
  - 32 B line size
  - 128 sets
- L2 cache
  - unified
  - 128 KB -- 2 MB

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



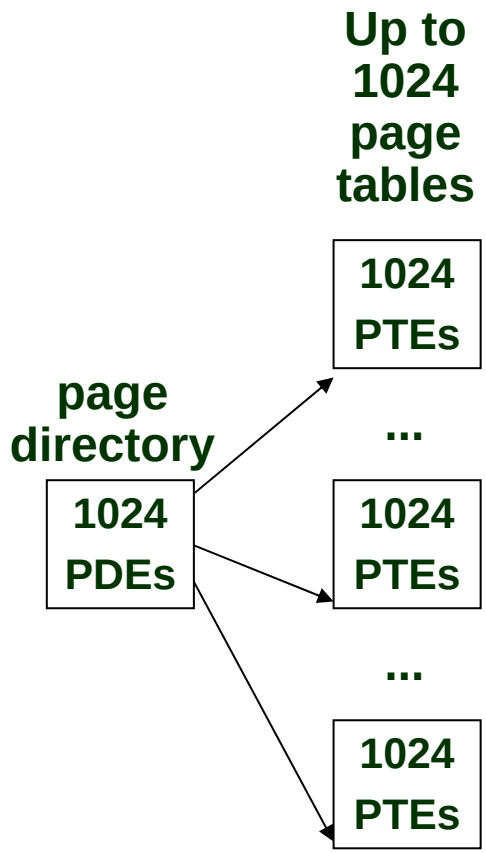
P6 2-level Page Table Structure

Page directory

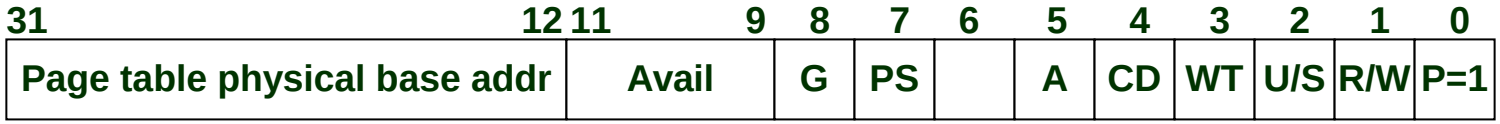
- 1024 4-byte page directory entries (PDEs) that point to page tables
- One page directory per process.
- Page directory must be in memory when its process is running
- Always pointed to by PDBR

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)



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)

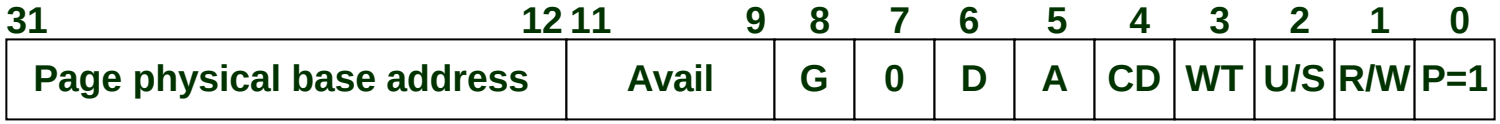
WT: write-through or write-back cache policy for this page table

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)

P6 Page Table Entry (PTE)



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

Avail: available for system programmers

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

D: dirty (set by MMU on writes)

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

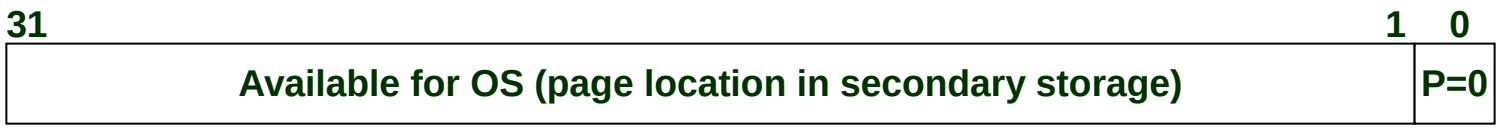
CD: cache disabled or enabled

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

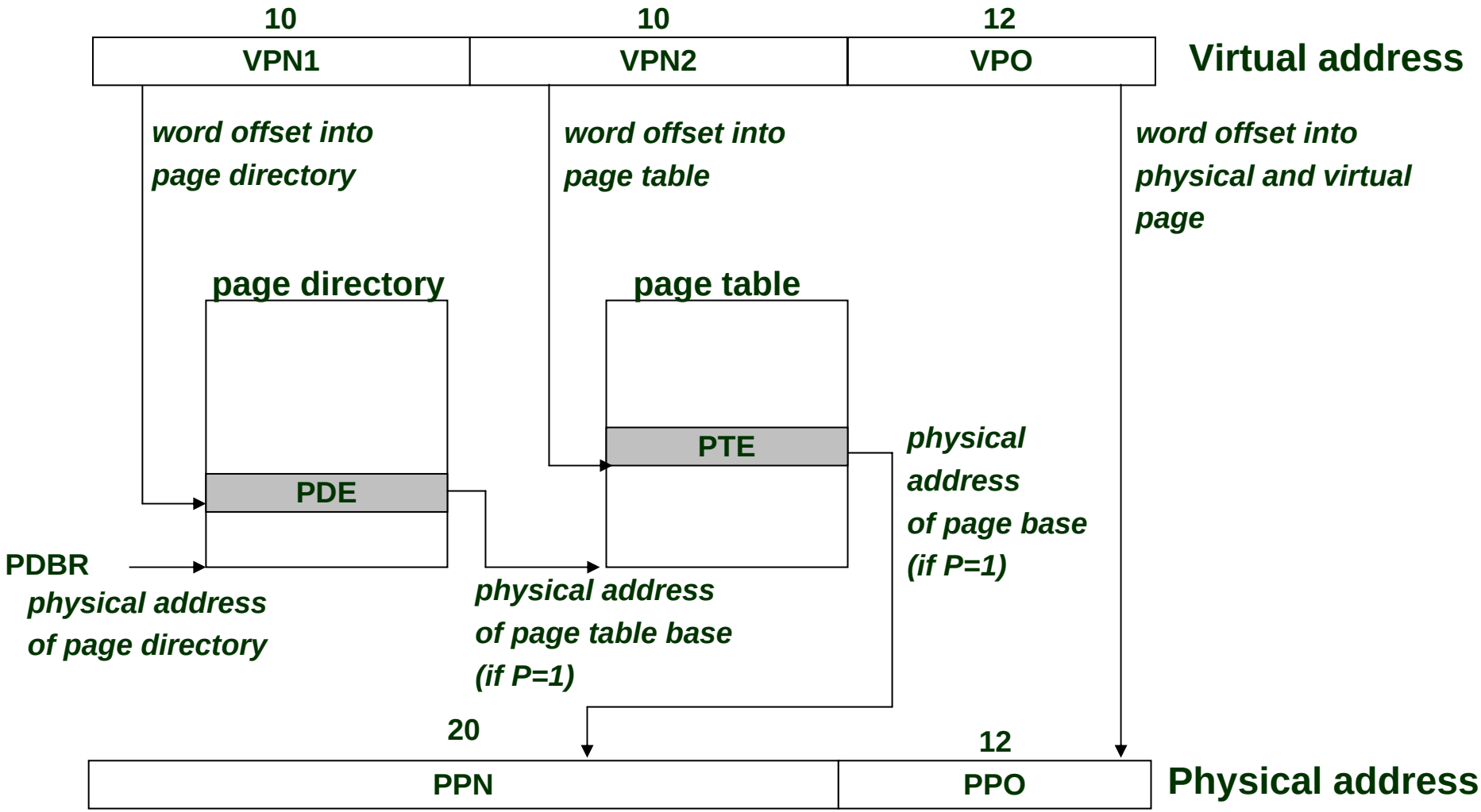
U/S: user/supervisor

R/W: read/write

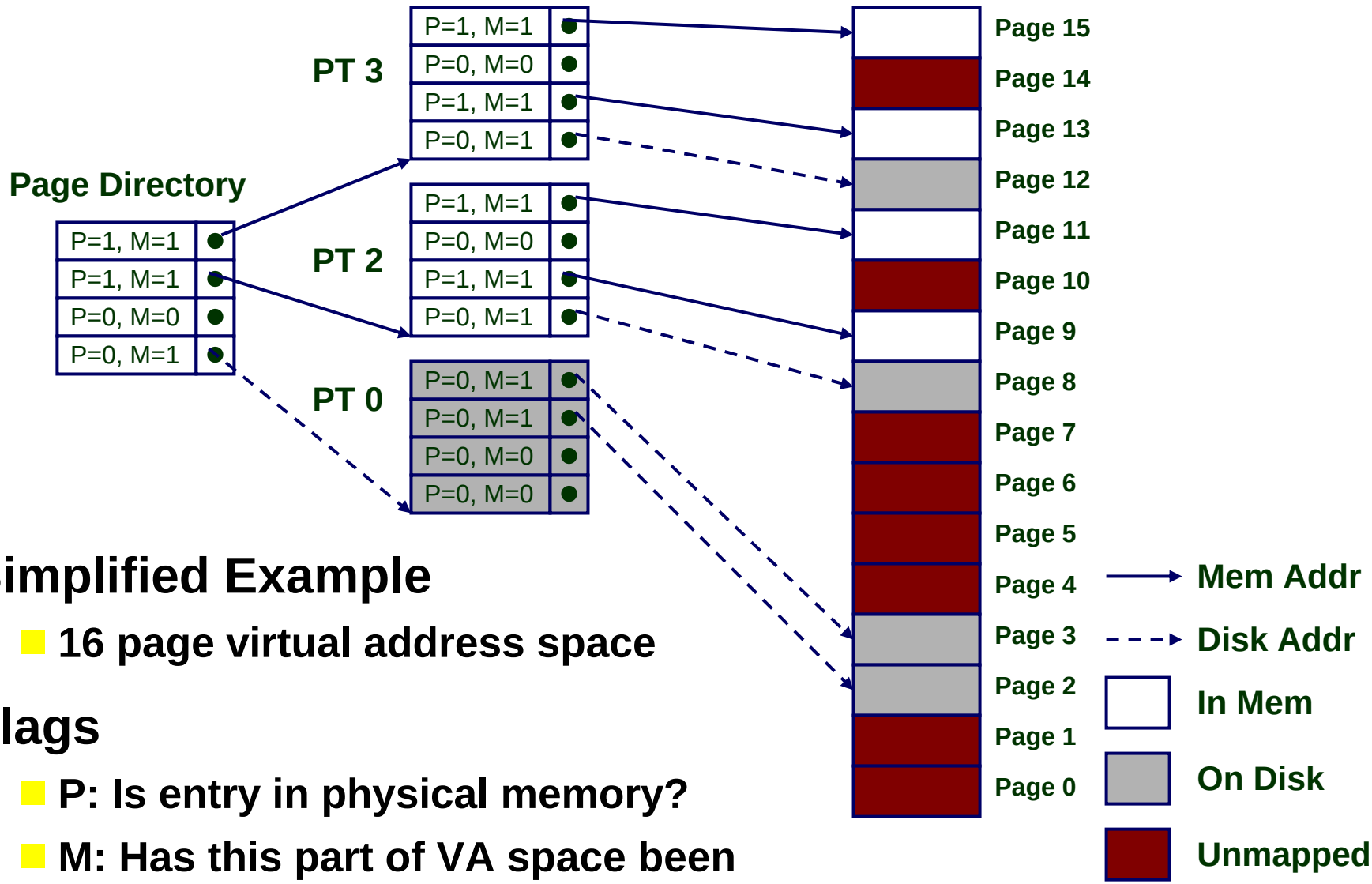
P: page is present in physical memory (1) or not (0)



How P6 Page Tables Map Virtual Addresses to Physical Ones



Representation of VM Address Space



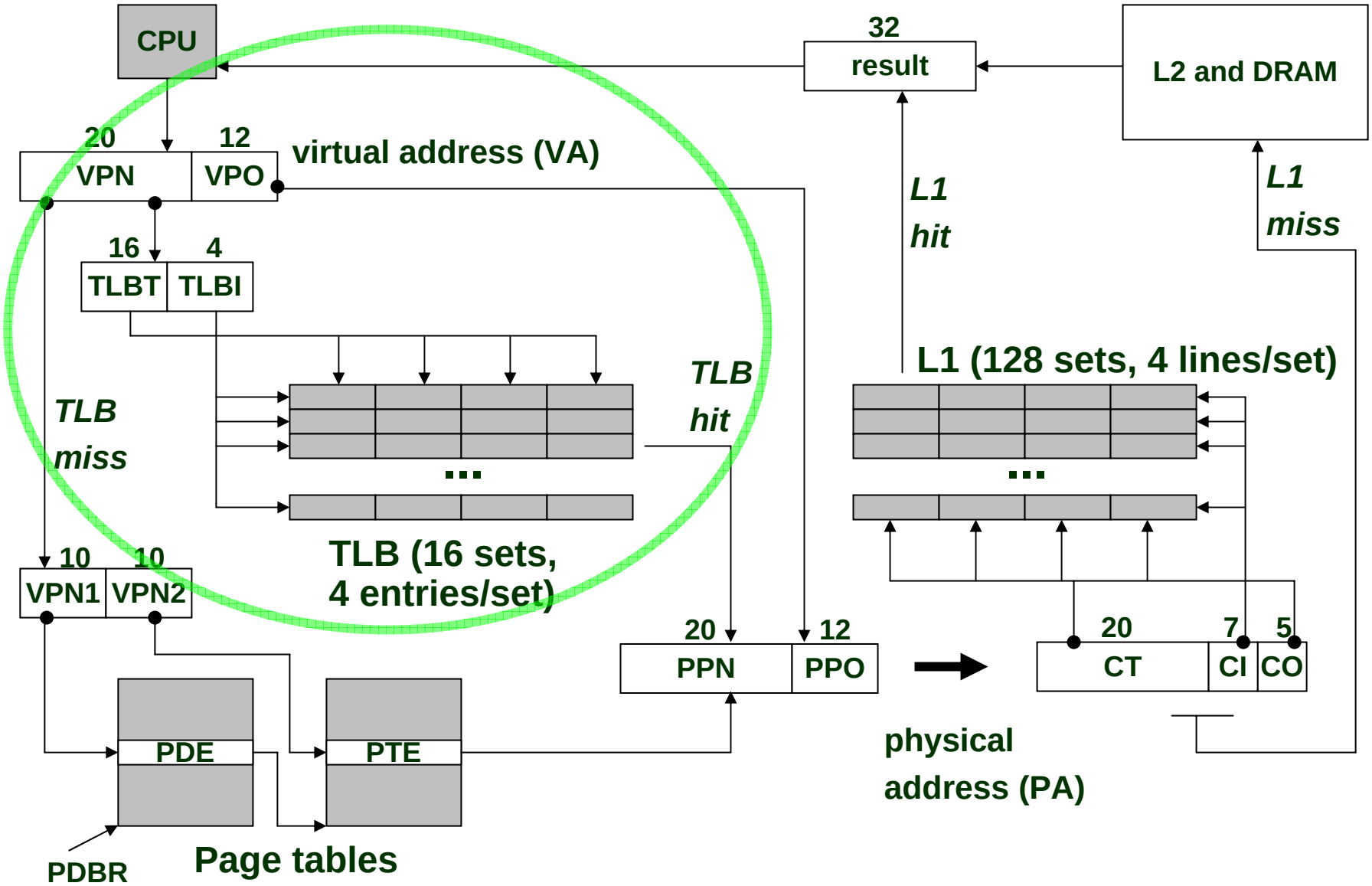
Simplified Example

- 16 page virtual address space

Flags

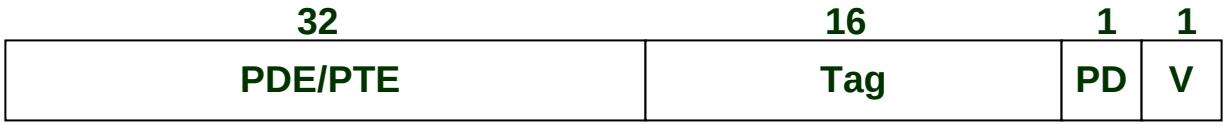
- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?

P6 TLB Translation



P6 TLB

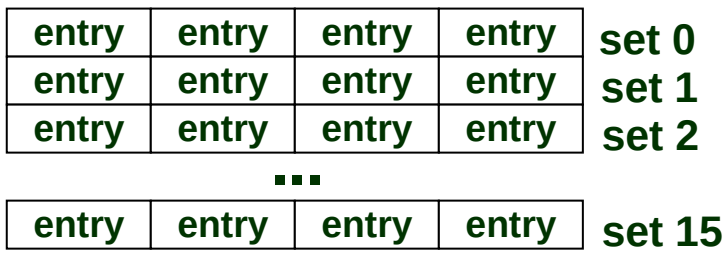
TLB entry (not all documented, so this is speculative):



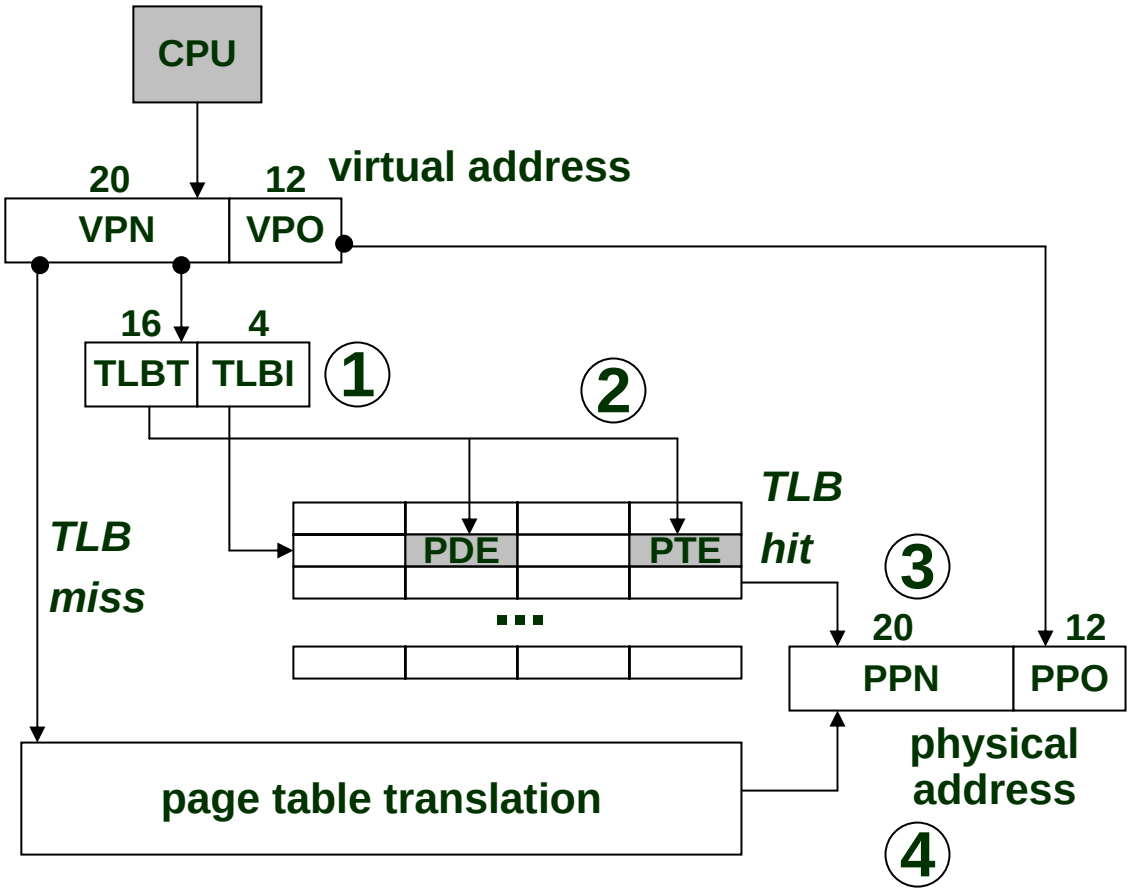
- **V: indicates a valid (1) or invalid (0) TLB entry**
- **PD: is this entry a PDE (1) or a PTE (0)?**
- **tag: disambiguates entries cached in the same set**
- **PDE/PTE: page directory or page table entry**

Structure of the data TLB:

- **16 sets, 4 entries/set**

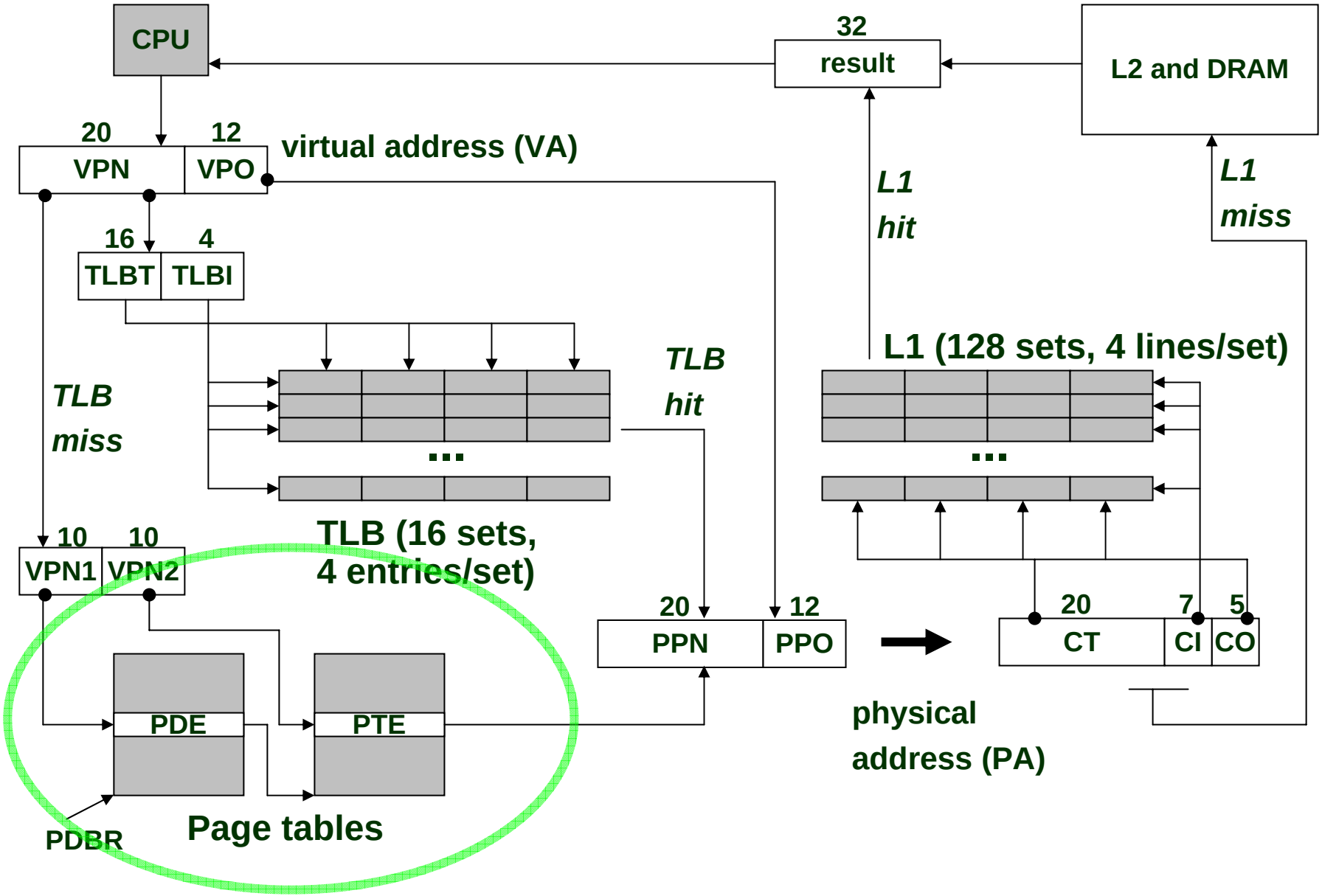


Translating with the P6 TLB

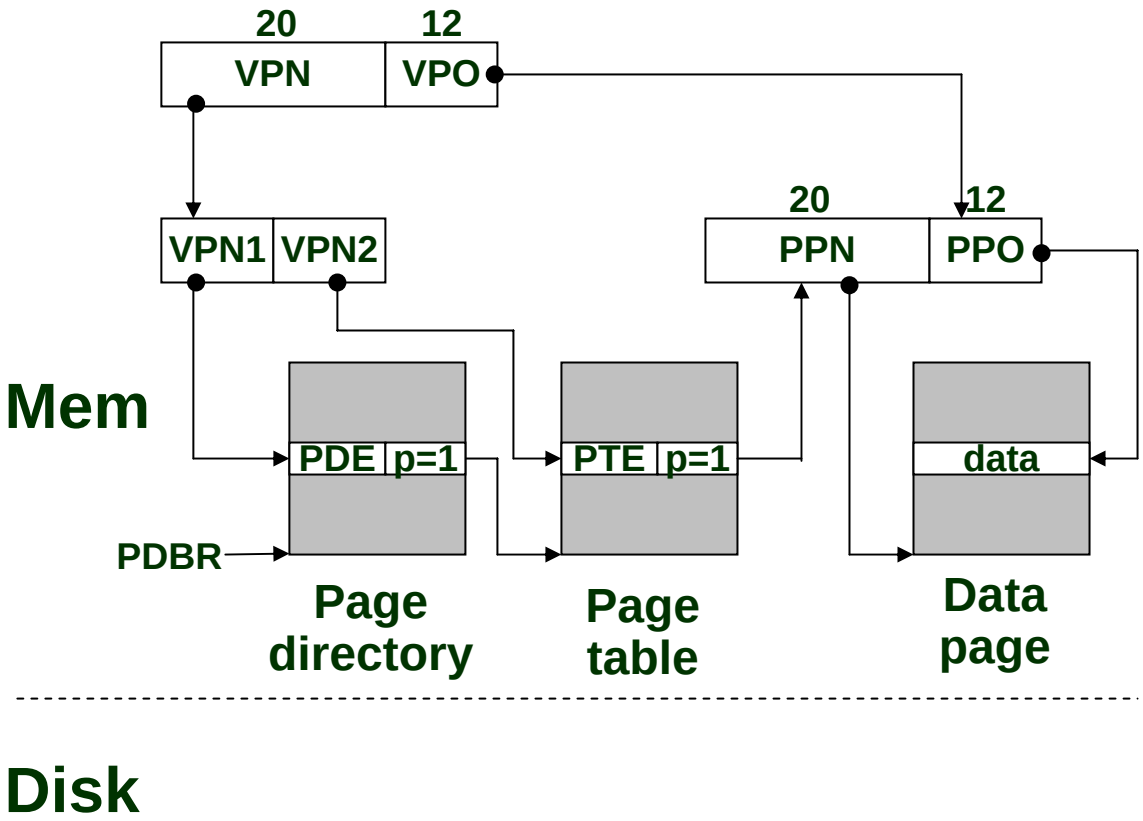


1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
   - 3. Yes: then build physical address.
4. No: then read PTE (and PDE if not cached) from memory and build physical address.

P6 Page Table Translation



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



Case 1/1: page table and page present.

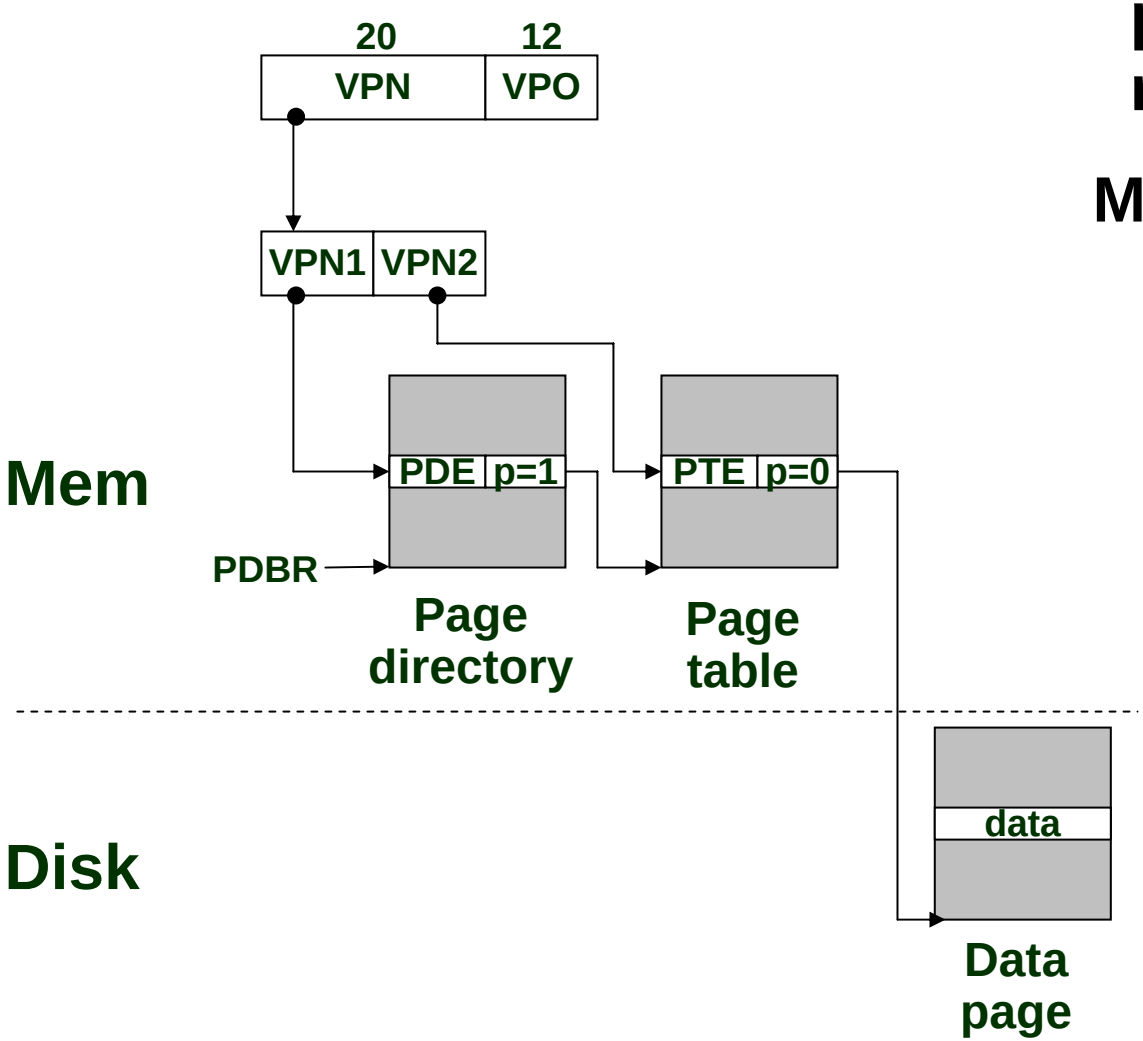
MMU Action:

- **MMU builds physical address and fetches data word.**

OS Action:

- **None**

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

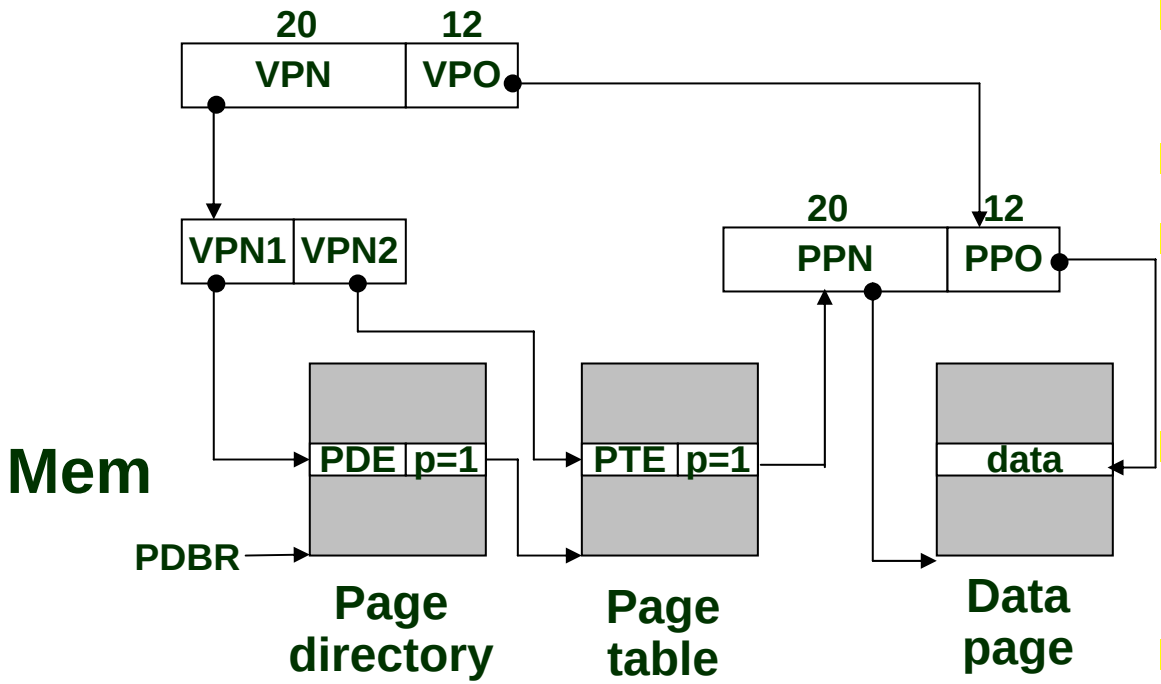


Case 1/0: page table present but page missing.

MMU Action:

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

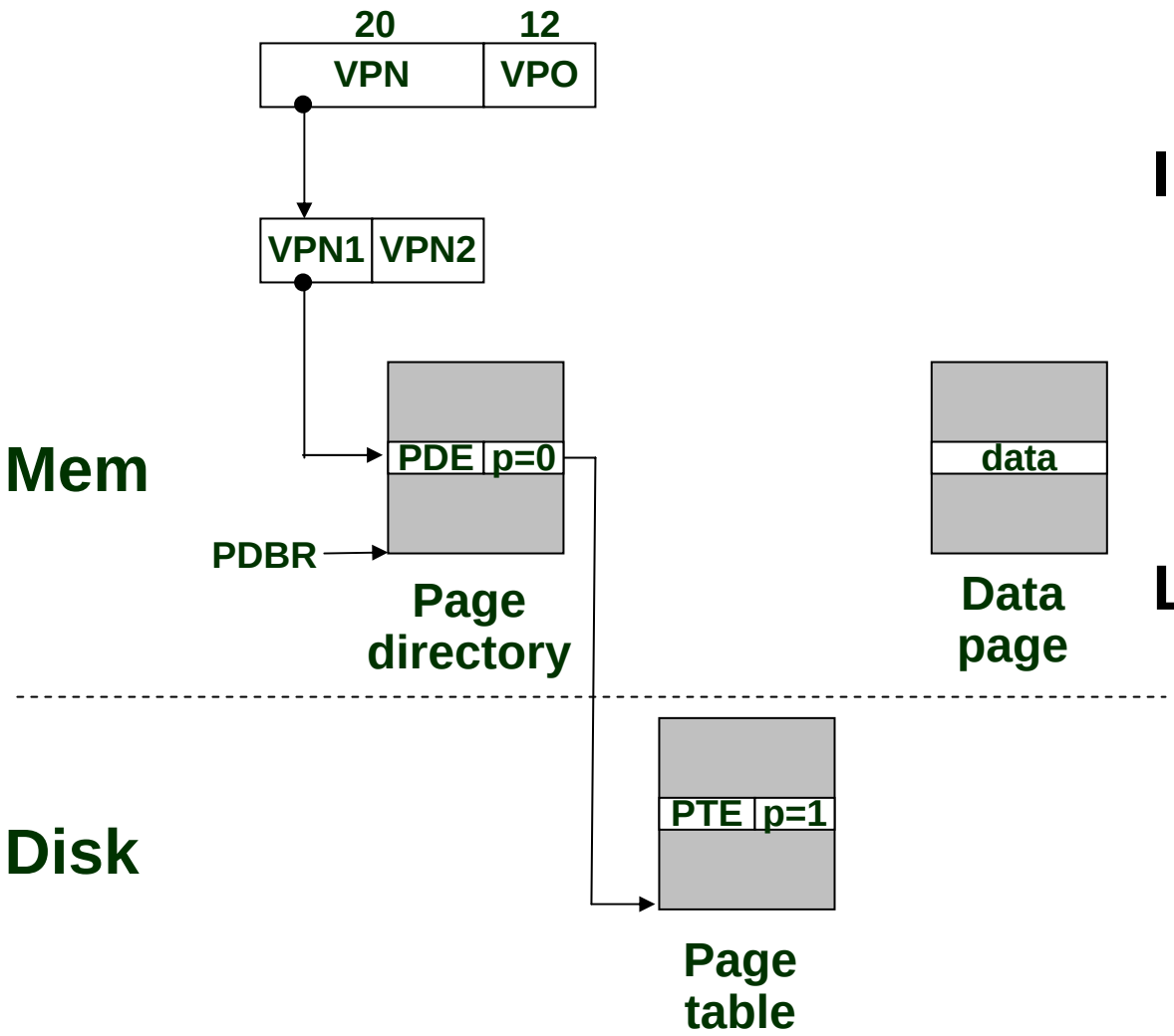
Translating with the P6 Page Tables (case 1/0, cont)



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 and copy to virtual page**
- **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.

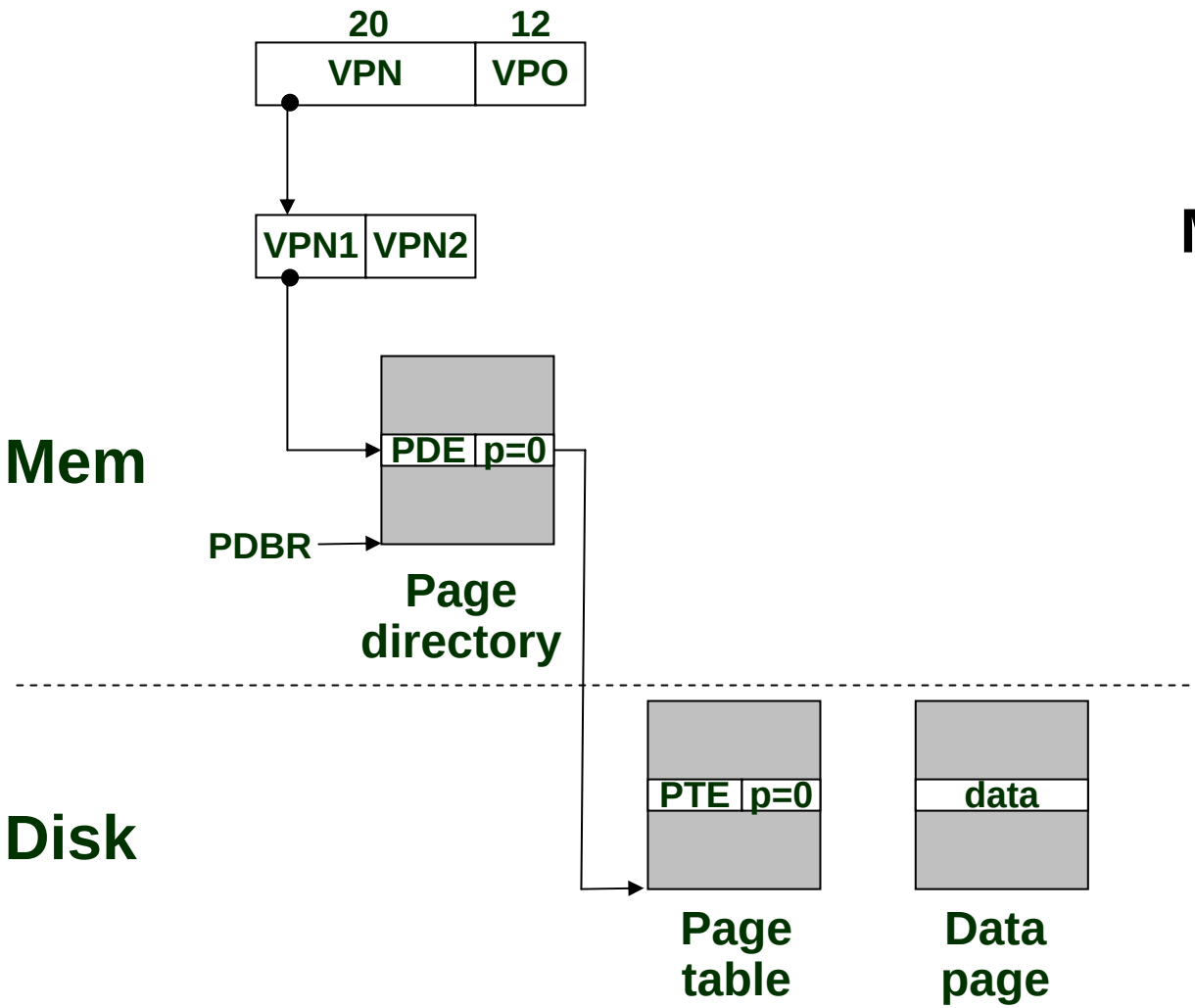
Introduces consistency issue.

- **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)

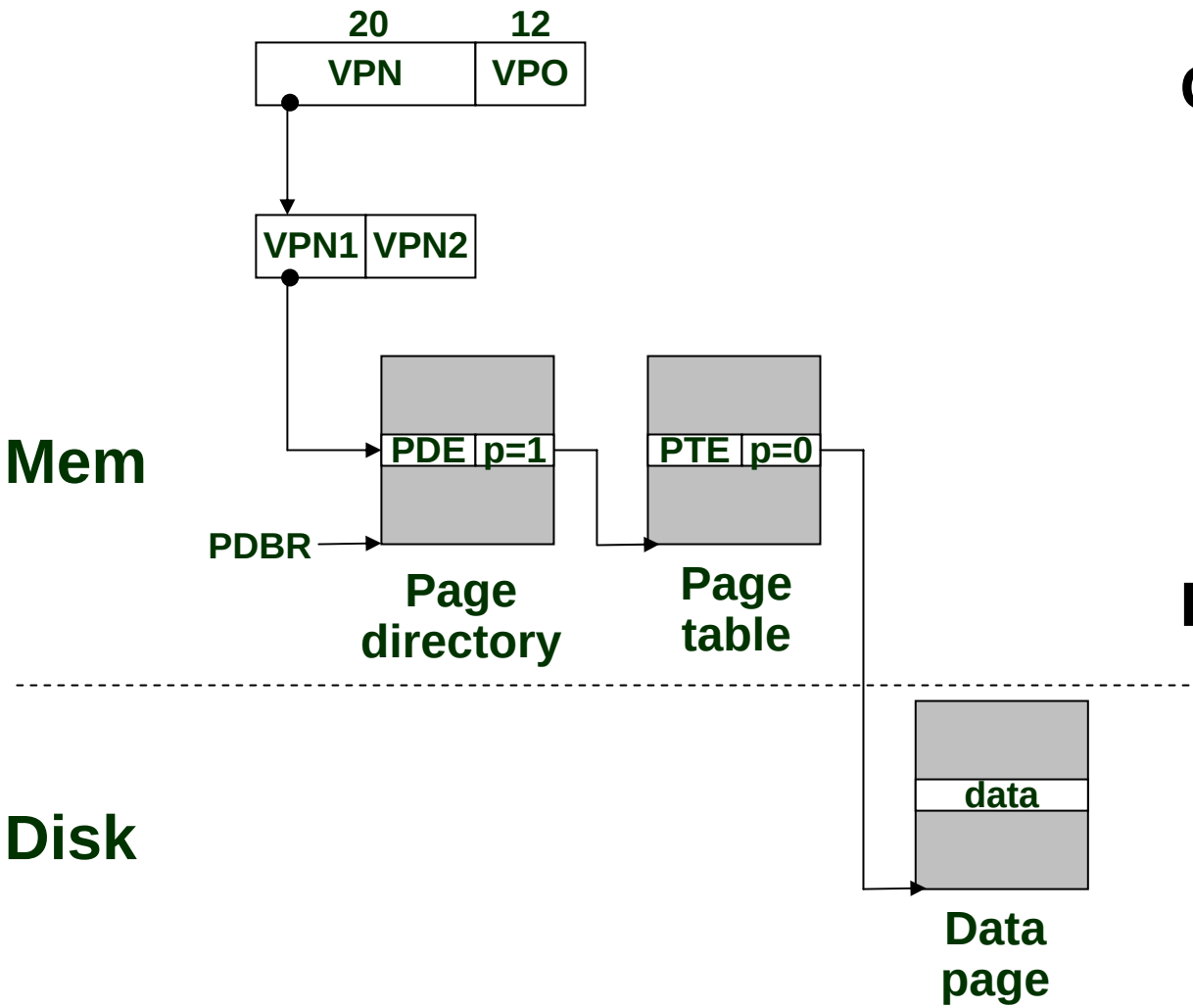


Case 0/0: page table and page missing.

MMU Action:

- **Page fault exception**

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

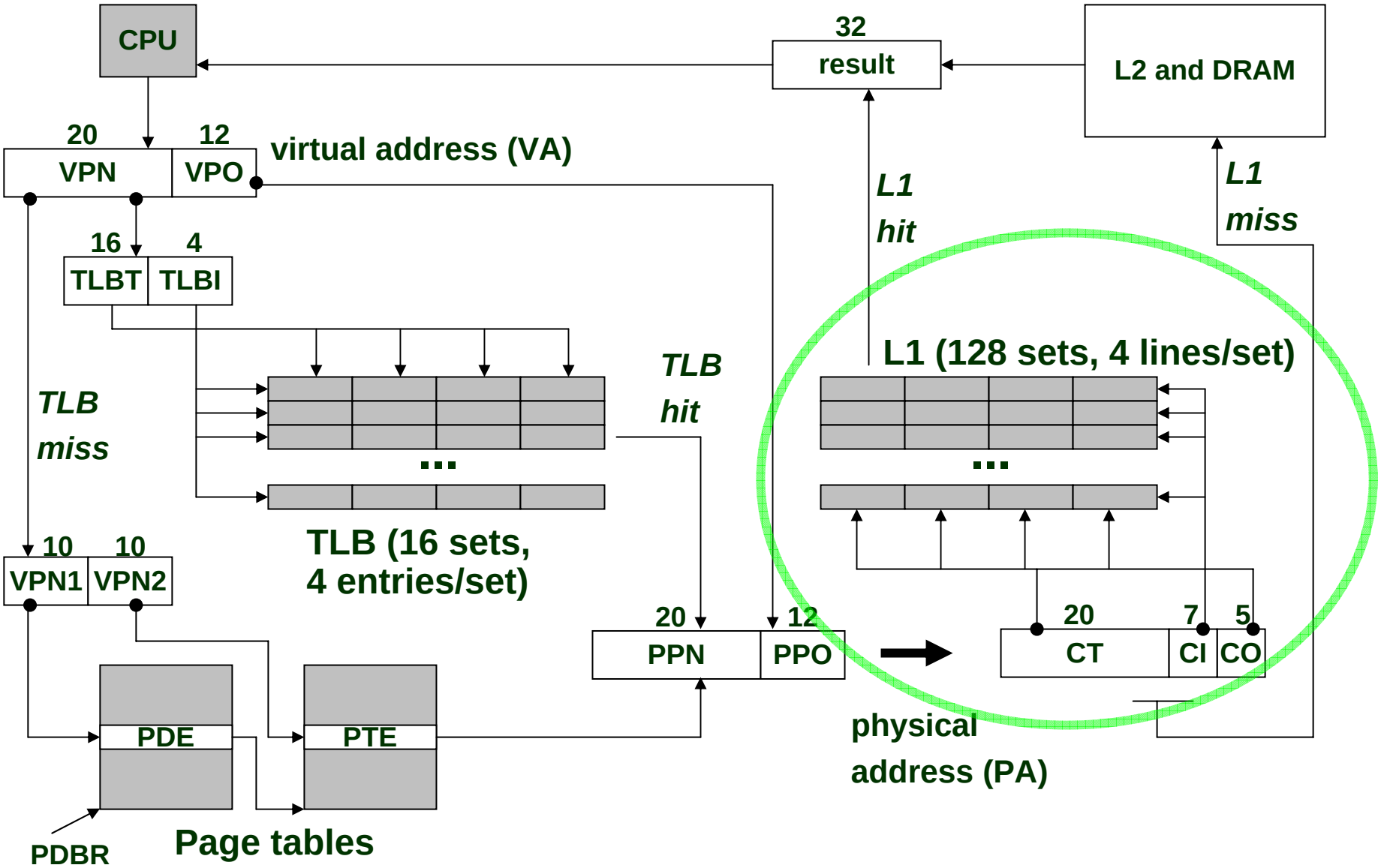


OS action:

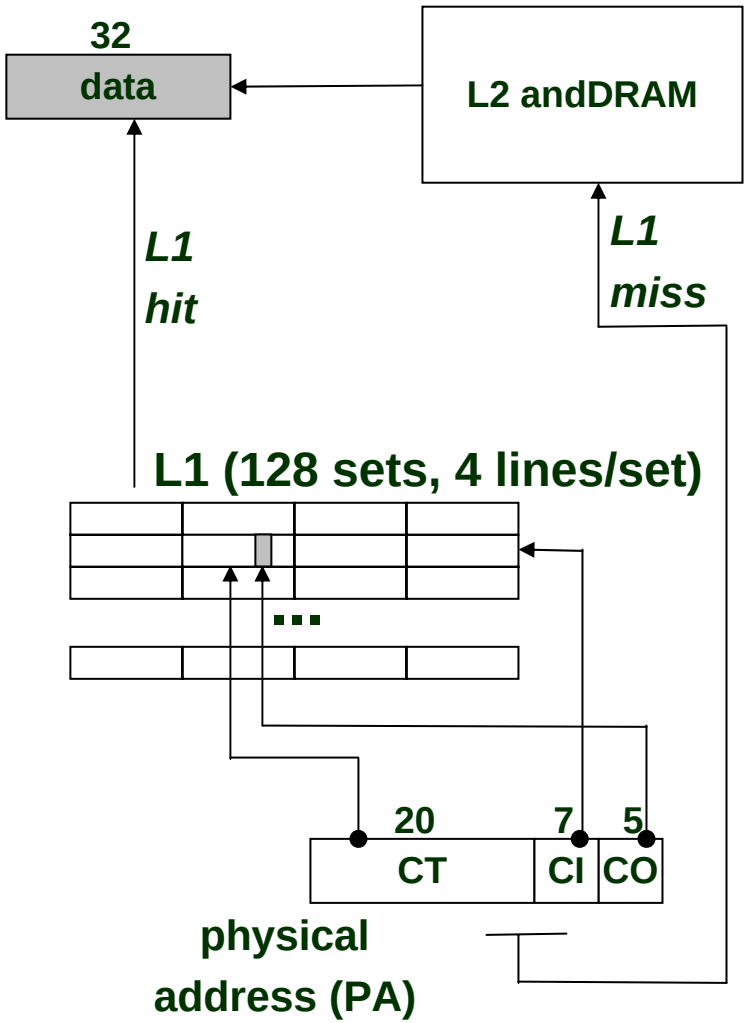
- **Swap in page table.**
- **Restart faulting instruction by returning from handler.**

Like case 0/1 from here on.

P6 L1 Cache Access



L1 Cache Access



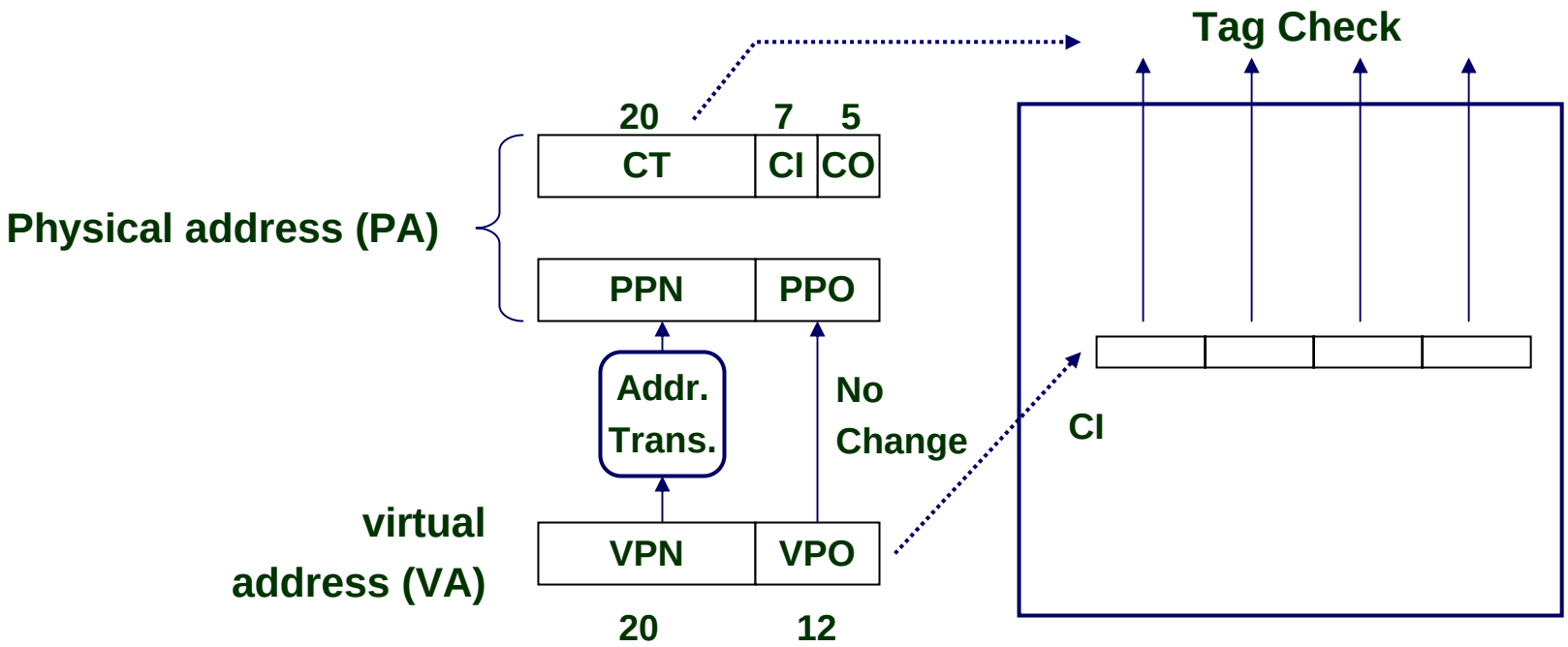
Partition physical address into CO, CI, and CT.

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
- Then check with CT from physical address
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

x86-64 Paging

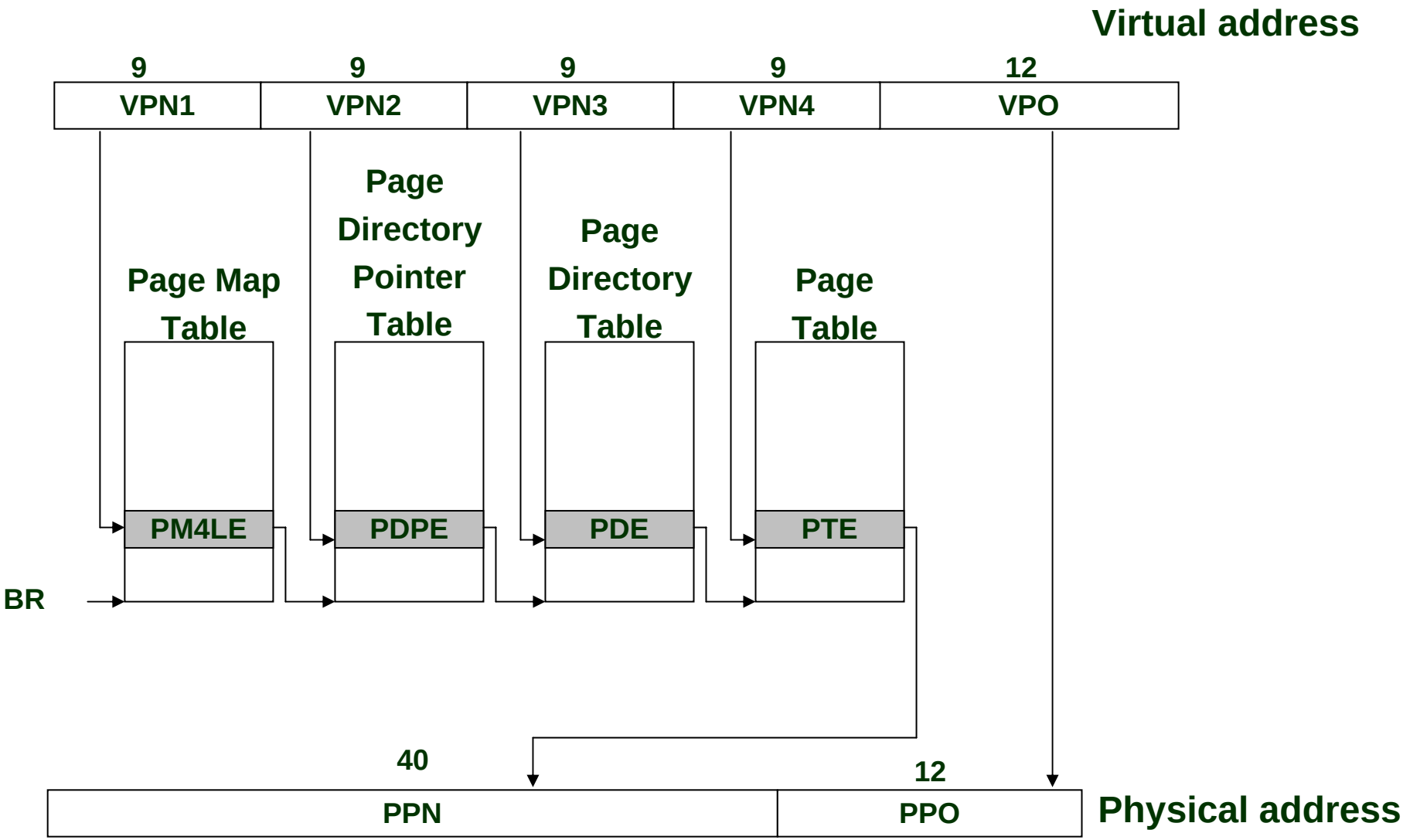
Origin

- AMD’s way of extending x86 to 64-bit instruction set
- Intel has followed with “EM64T”

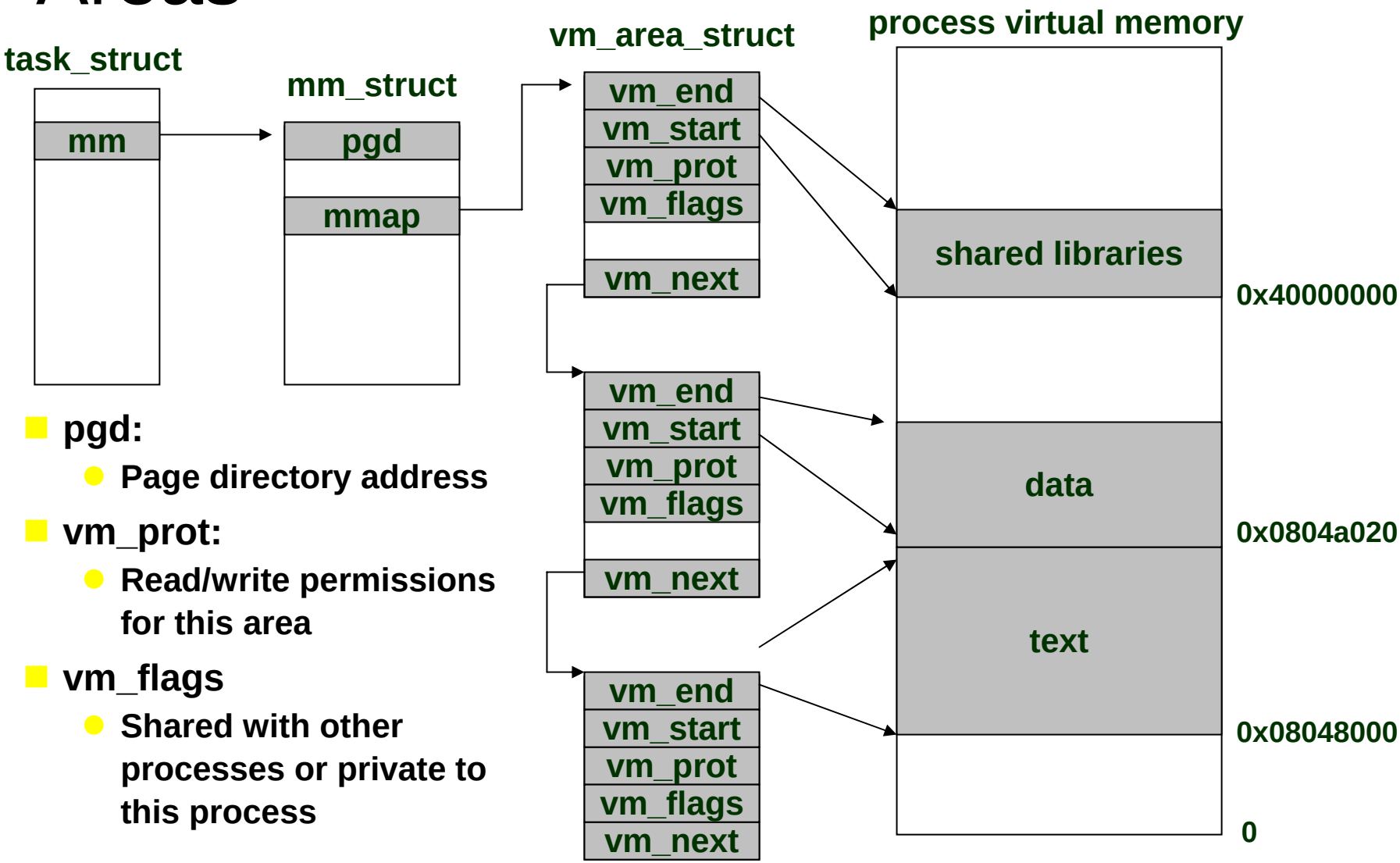
Requirements

- 48 bit virtual address
  - 256 terabytes (TB)
  - Not yet ready for full 64 bits
- 52 bit physical address
  - Requires 64-bit table entries
- 4KB page size
  - Only 512 entries per page

x86-64 Paging

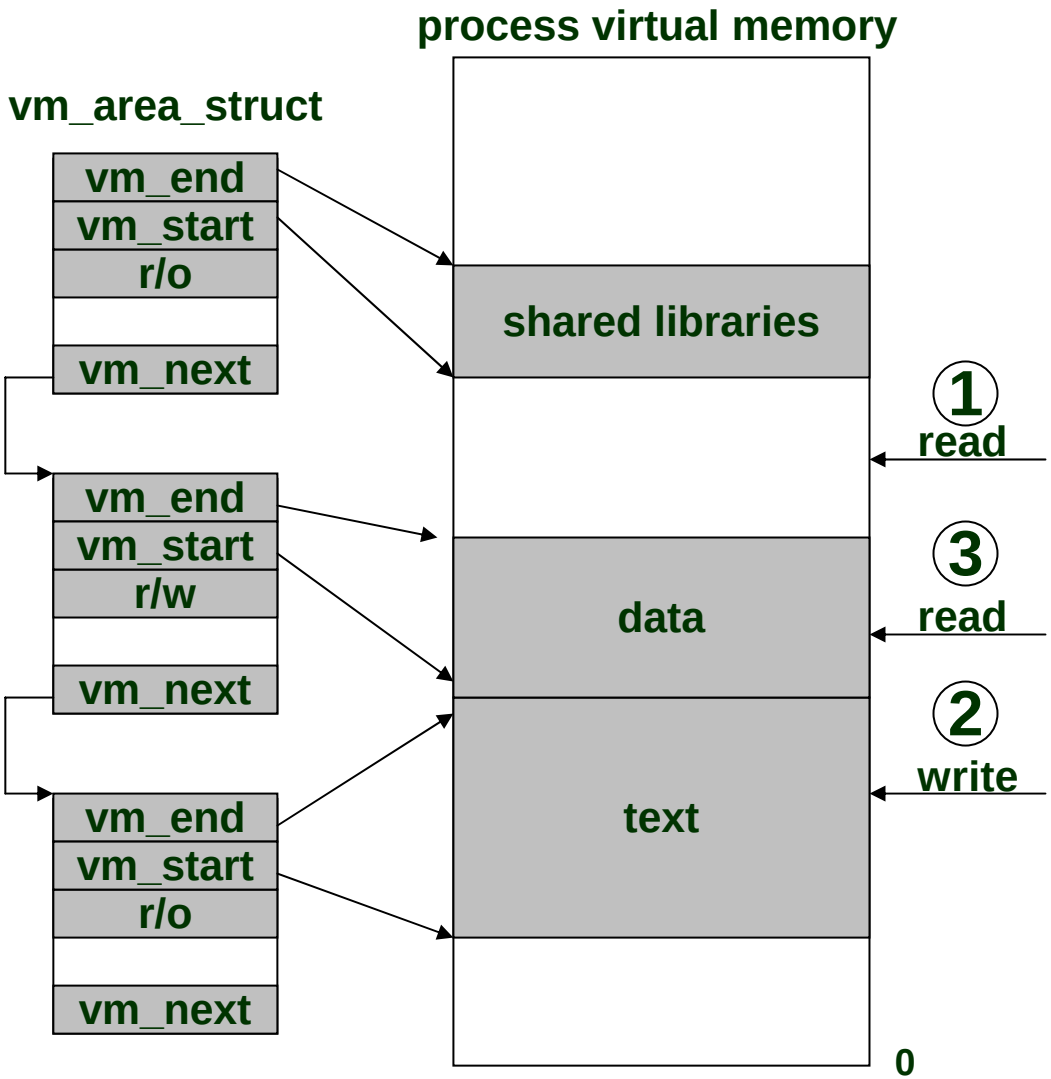


Linux Organizes VM as Collection of "Areas"



- **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 then signal segmentation violation (e.g. (1))

Is the operation legal?

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

If OK, handle fault

- e.g., (3)

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)**
    - » **Initial page bytes are zeros**
- **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)
```

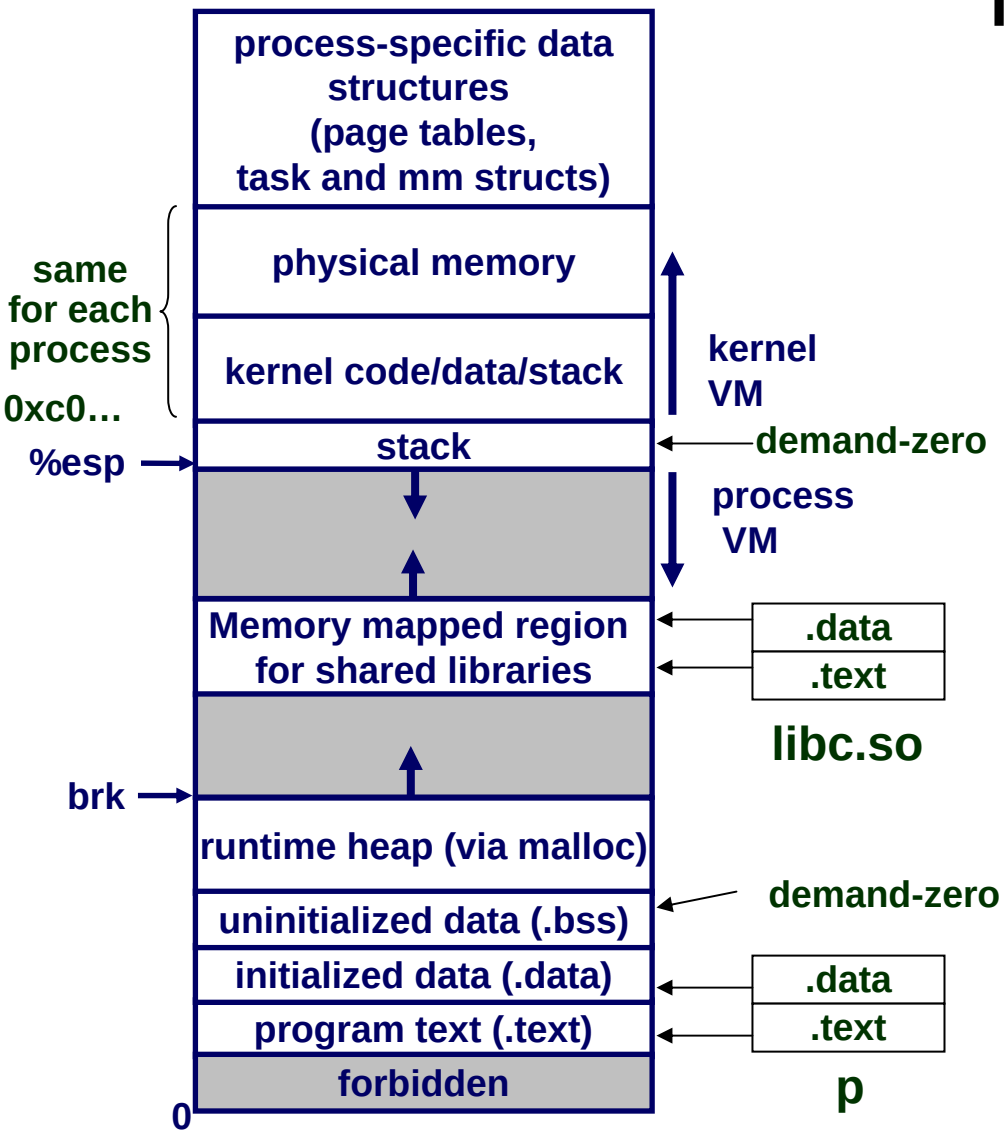
- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually `0` for don't care).
  - `prot`: MAP_READ, MAP_WRITE
  - `flags`: MAP_PRIVATE, MAP_SHARED
- Return a pointer to the mapped area.
- Example: fast file copy
  - Useful for 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 itself 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);
}
```

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 swap in code and 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 are sharing all of their pages.
  - How to get separate spaces without copying all the virtual pages from one space to another?
    - » "copy on write" technique.
- Copy-on-write
  - Make pages of writeable areas read-only
  - Flag vm_area_struct's for these areas as private "copy-on-write".
  - Writes by either process to these pages will cause page faults.
    - » 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

Cache Memory

- 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
- Combination of hardware & software implementation
  - Software management of tables, allocations
  - Hardware access of tables
  - Hardware caching of table entries (TLB)

Next Time

- **RCX Hardware/Peripherals**
- **BrickOS real-time operating system**
  - **Successor to legOS 0.2.4**
  - **Available at: http://brickos.sourceforge.net**
  - **Installation/building using GCC cross-compiler**
- **BrickOS Kernel**
  - **Task management**
  - **Memory management**
  - **Interprocess communicaton**
  - **IR networking**

RCX Hardware

Hitachi H8/3292 microcontroller

16-bit Timer

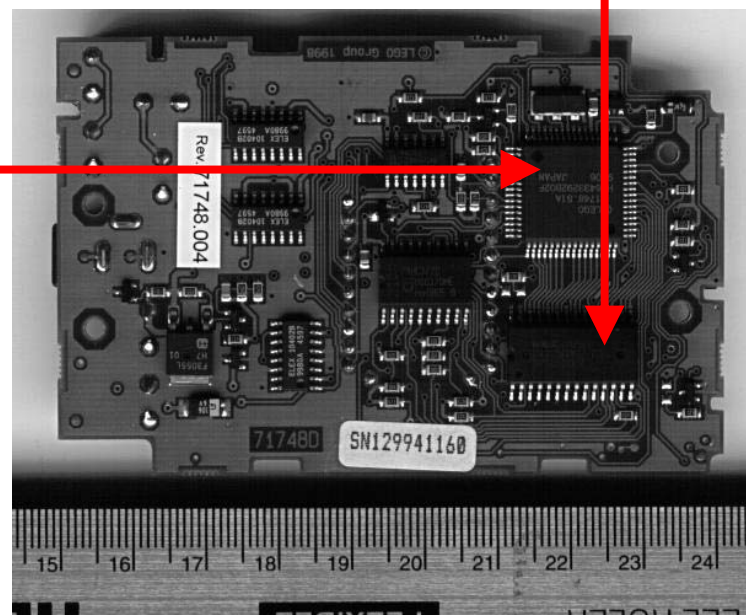
8-bit Timer

I/O Ports

16k ROM

A/D Converter

32k RAM



BrickOS 0.2.6.10 Installation

http://brickos.sourceforge.net/documents.htm

Windows XP Installation:

- **Install Cygwin**
- **Build the Hitachi-H8 cross compiler (h8300-hitachi-hms-gcc.exe, ..)**
- **Install the brickOS source code and build kernel (brickOS.srec)**
- **Try it:**
  - **Download the kernel: $ ./firmdl3 ../boot/brickOS.srec**
  - **Download a sample application: $ ./dll ../demo/sound.lx**

Linux Installation is similar – brickOS is installed in the Linux Lab.