Virtual Memory

Alan L. Cox
alc@rice.edu

Some slides adapted from CMU 15.213 slides

Objectives

Be able to explain the rationale for virtual memory (VM)

Be able to describe the functionality provided by VM

Be able to translate virtual addresses to physical addresses

Be able to explain how VM benefits fork() and execve()

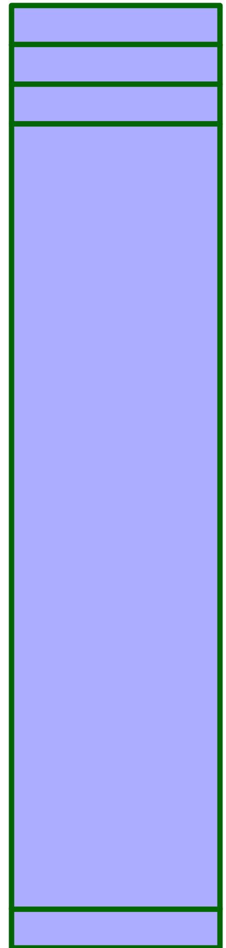
Virtual Memory

Programs refer to virtual memory addresses

- `movl (%ecx),%eax`
- Conceptually very large array of bytes
- Each byte has its own address
- System provides address space private to particular “process”

Compiler and run-time system allocate VM

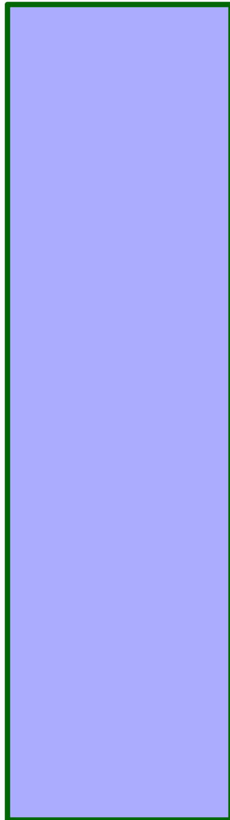
- Where different program objects should be stored
- All allocation within single virtual address space

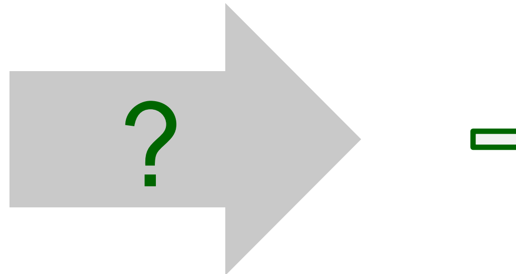


Problem 1: How Does Everything Fit?

64-bit addresses:
16 Exabyte (16 billion GB!)

Physical main memory:
Tens or Hundreds of Gigabytes





And there are many processes ….

Problem 2: Memory Management

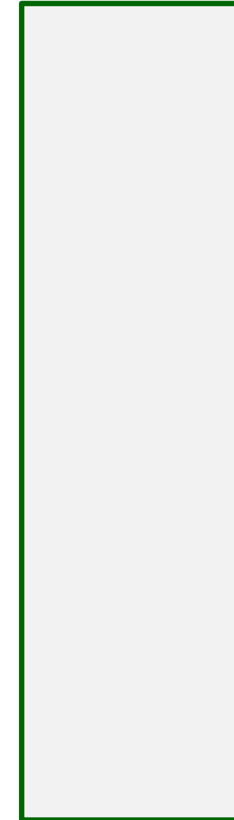
Physical main memory

Process 1
Process 2
Process 3
…
Process n

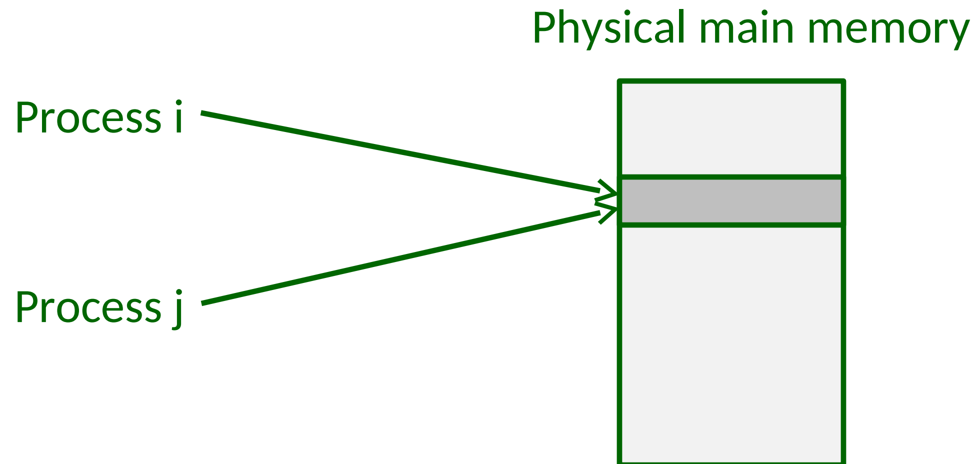
X

stack
heap
`.text`
`.data`
…

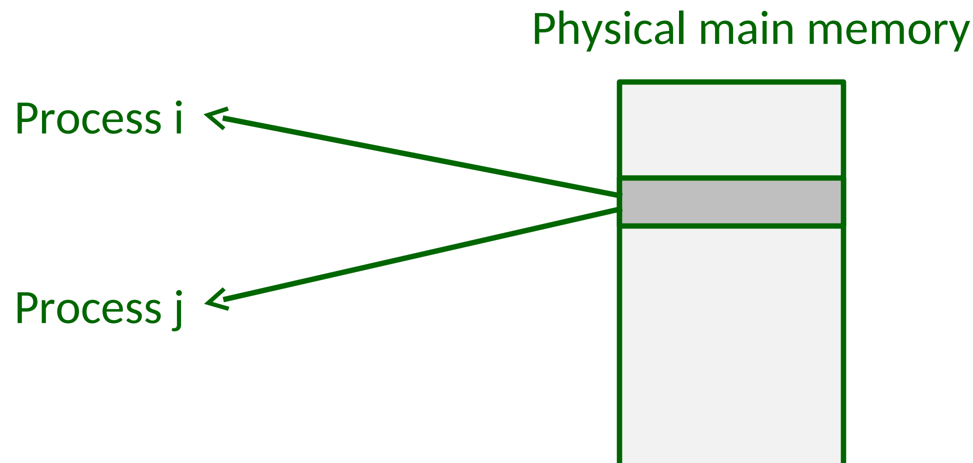
What goes where?



Problem 3: How To Protect

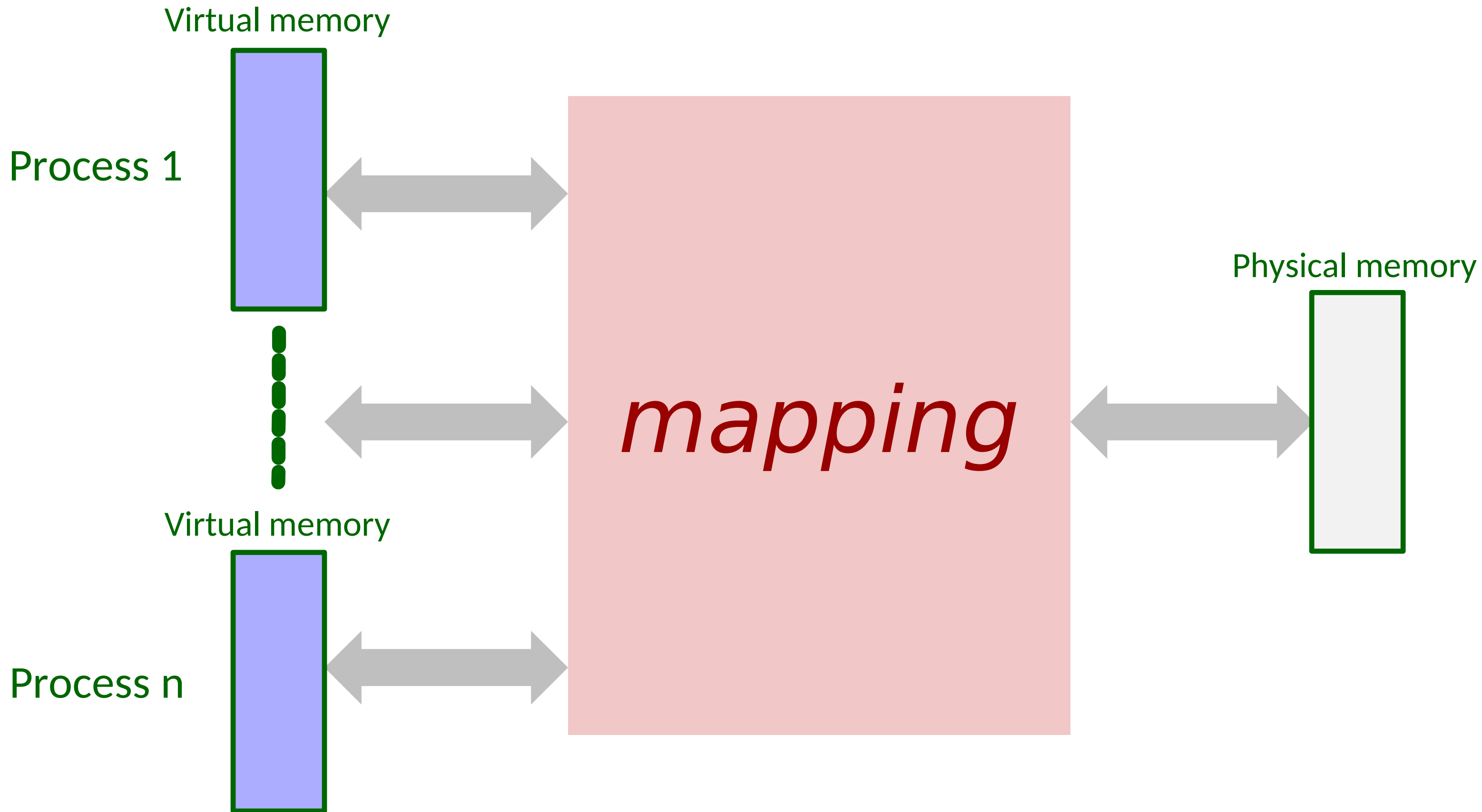


Problem 4: How To Share?



Solution: Indirection

"All problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection." – David Wheeler



Mapping solves the previous problems

Each process gets its own private memory space

Address Spaces

Virtual address space: Set of $N = 2^n$ virtual addresses
{0, 1, 2, 3, …, N-1}

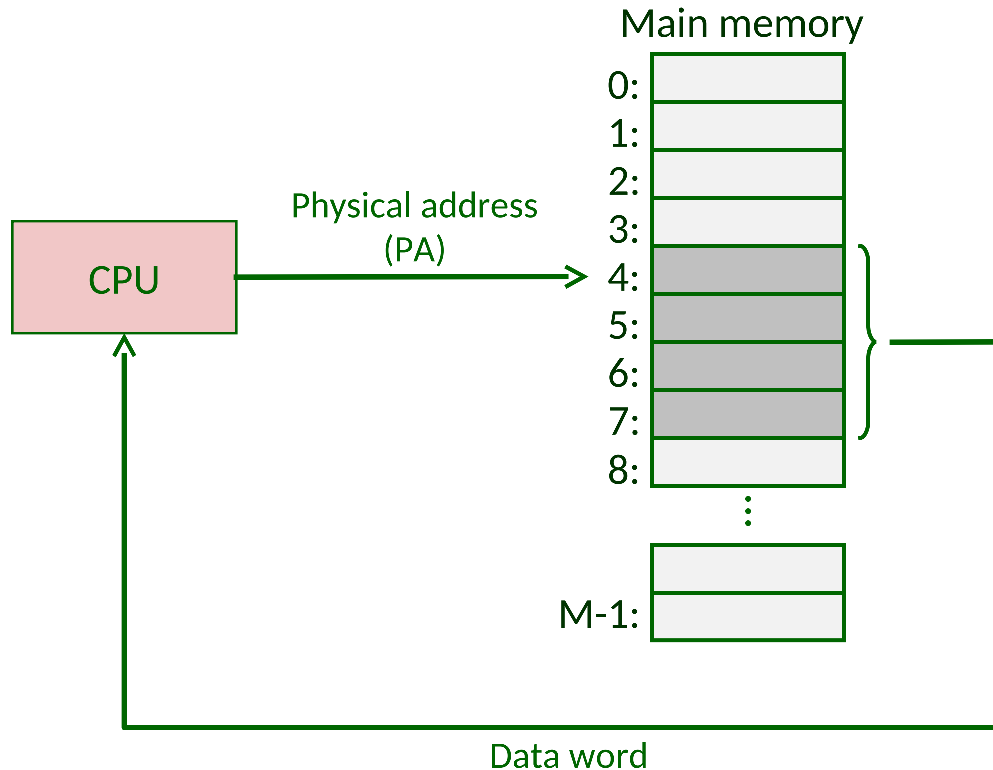
Physical address space: Set of $M = 2^m$ physical addresses
{0, 1, 2, 3, …, M-1}

Clear distinction between data (bytes) and their attributes (addresses)

Each data object can have multiple addresses

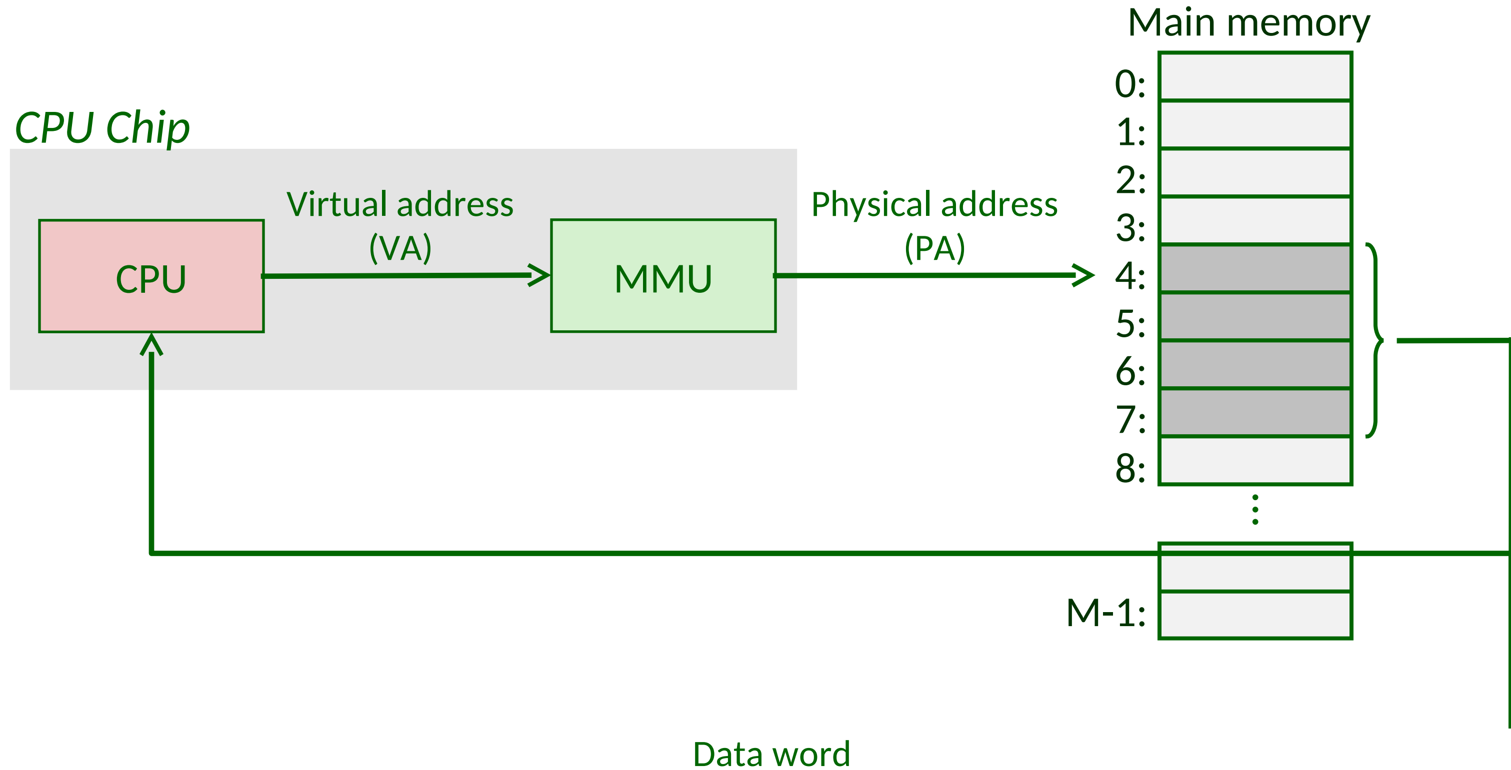
Every byte in main memory:
one physical address, one (or more) virtual addresses

A System Using Physical Addressing



Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



Used in all modern servers, desktops, laptops, tablets, and cell phones

Why Virtual Memory (VM)?

Efficient use of limited main memory (RAM)

- Use RAM as a cache for parts of a virtual address space
  - some non-cached parts stored on disk
  - some (unallocated) non-cached parts stored nowhere
- Keep only active areas of virtual address space in RAM
  - transfer data back and forth to disk as needed
- Share immutable code and data

Simplifies memory management for programmers

- Each process gets a full private address space

Isolates address spaces

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

Outline

Virtual memory (VM)

- Overview and motivation
- VM as tool for caching
- VM as tool for memory management
- VM as tool for memory protection
- Address translation
- mmap(), fork(), exec()

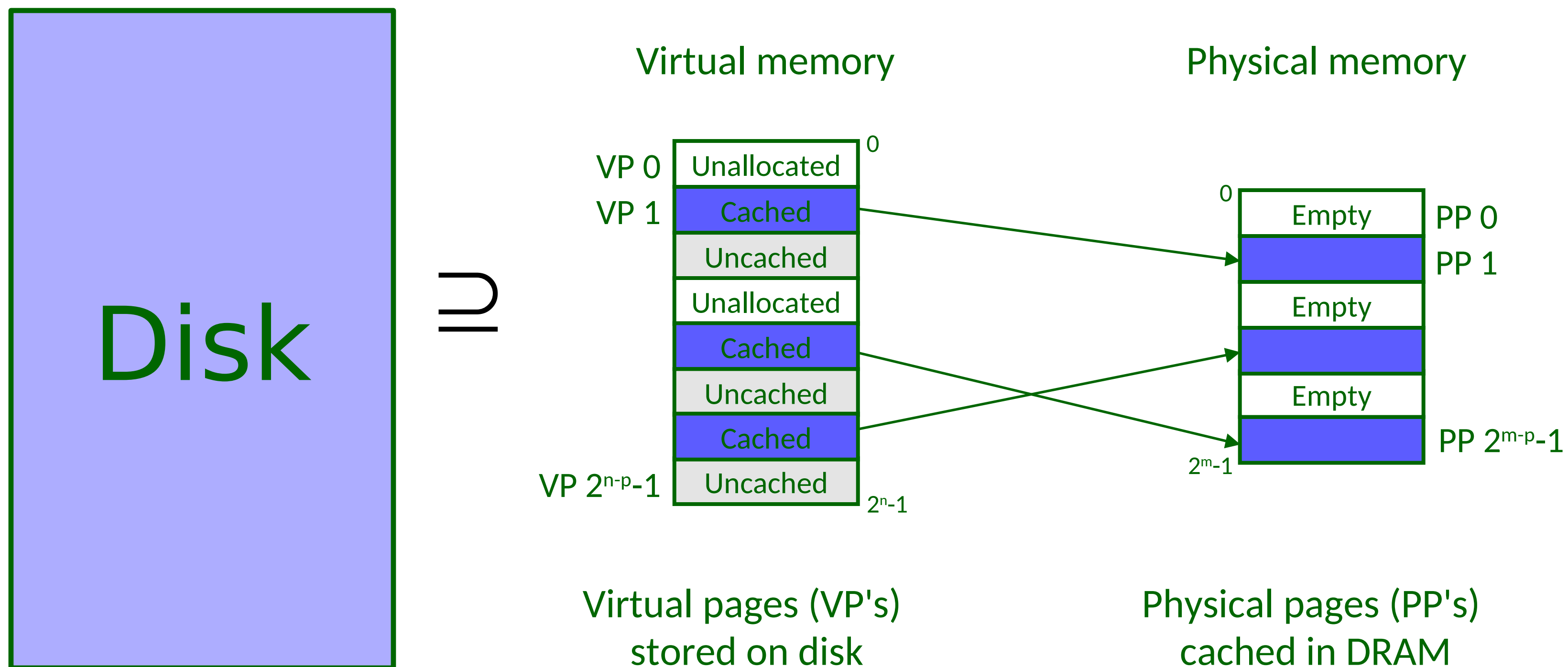
VM as a Tool for Caching

Virtual memory: **array of N = 2^n contiguous bytes**

- **think of the array (allocated part) as being stored on disk**

Physical main memory (DRAM) = cache for allocated virtual memory

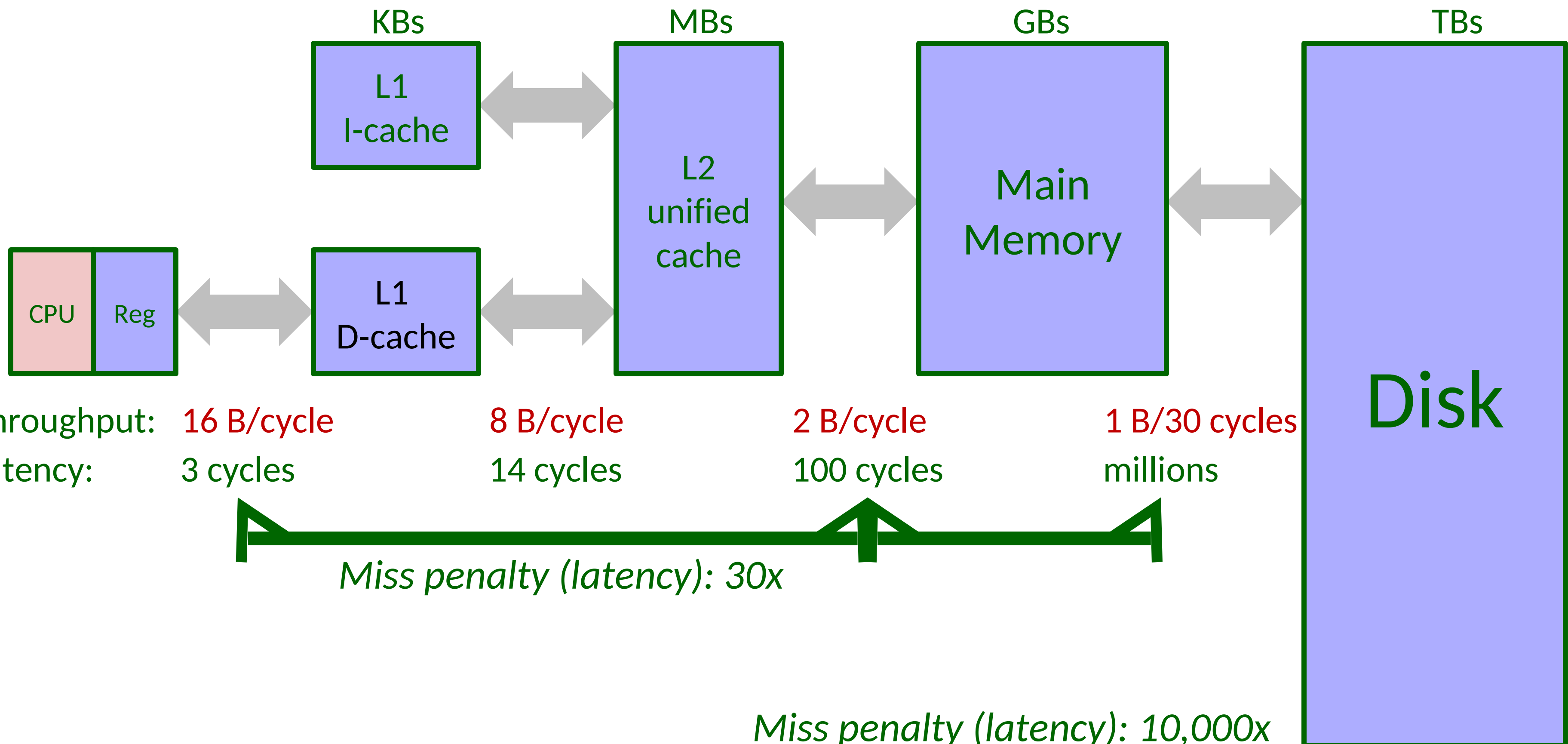
Basic unit: page; size = 2^p



An Example Memory Hierarchy

Not drawn to scale

L1/L2 cache: 64 B blocks



DRAM Cache Organization

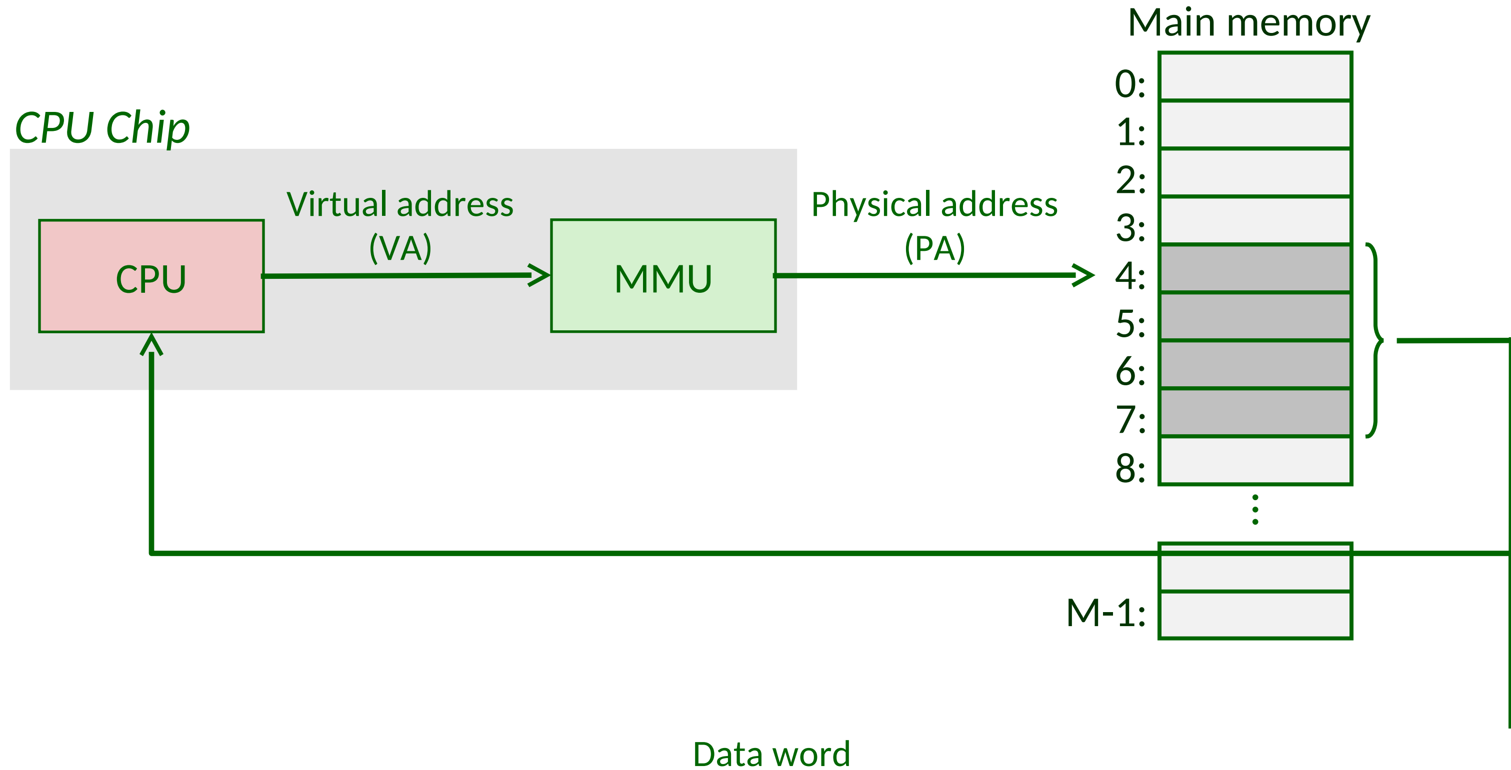
DRAM cache organization driven by the enormous miss penalty

- **DRAM is about *10x* slower than the SRAM used to implement CPU caches**
- **A solid-state disk is about *100x* slower than DRAM and a mechanical disk is about *10,000x* slower than DRAM**
  - **For first byte, faster for next byte**

Consequences

- **Large page size: at least 4-8 KB, sometimes 2-4 MB**
- **Fully associative**
  - **Any VP can be placed in any PP**
- **Highly sophisticated replacement algorithms**
  - **Too complicated and open-ended to be implemented in hardware**
- **Write-back rather than write-through**

A System Using Virtual Addressing

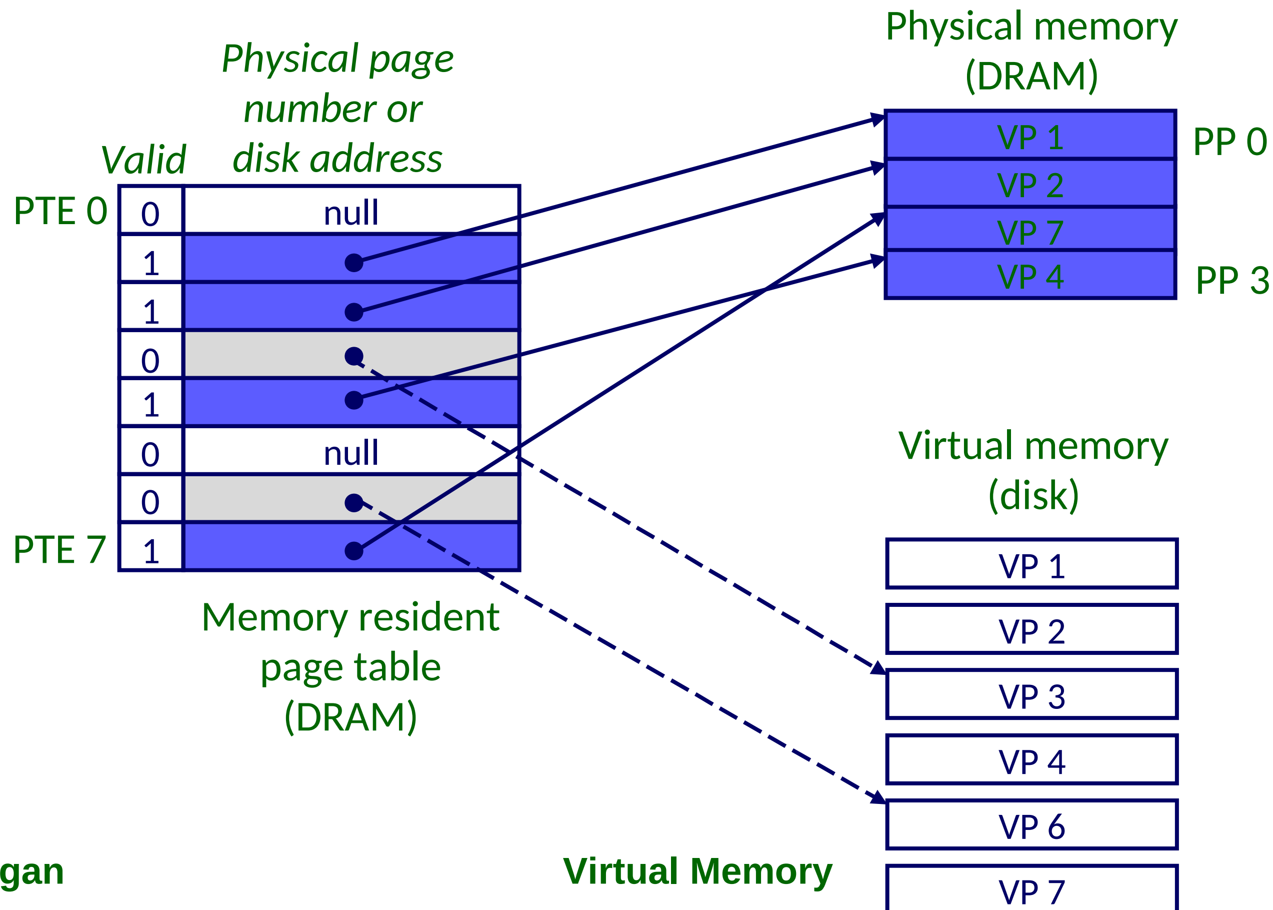


Used in all modern servers, laptops, and cell phones

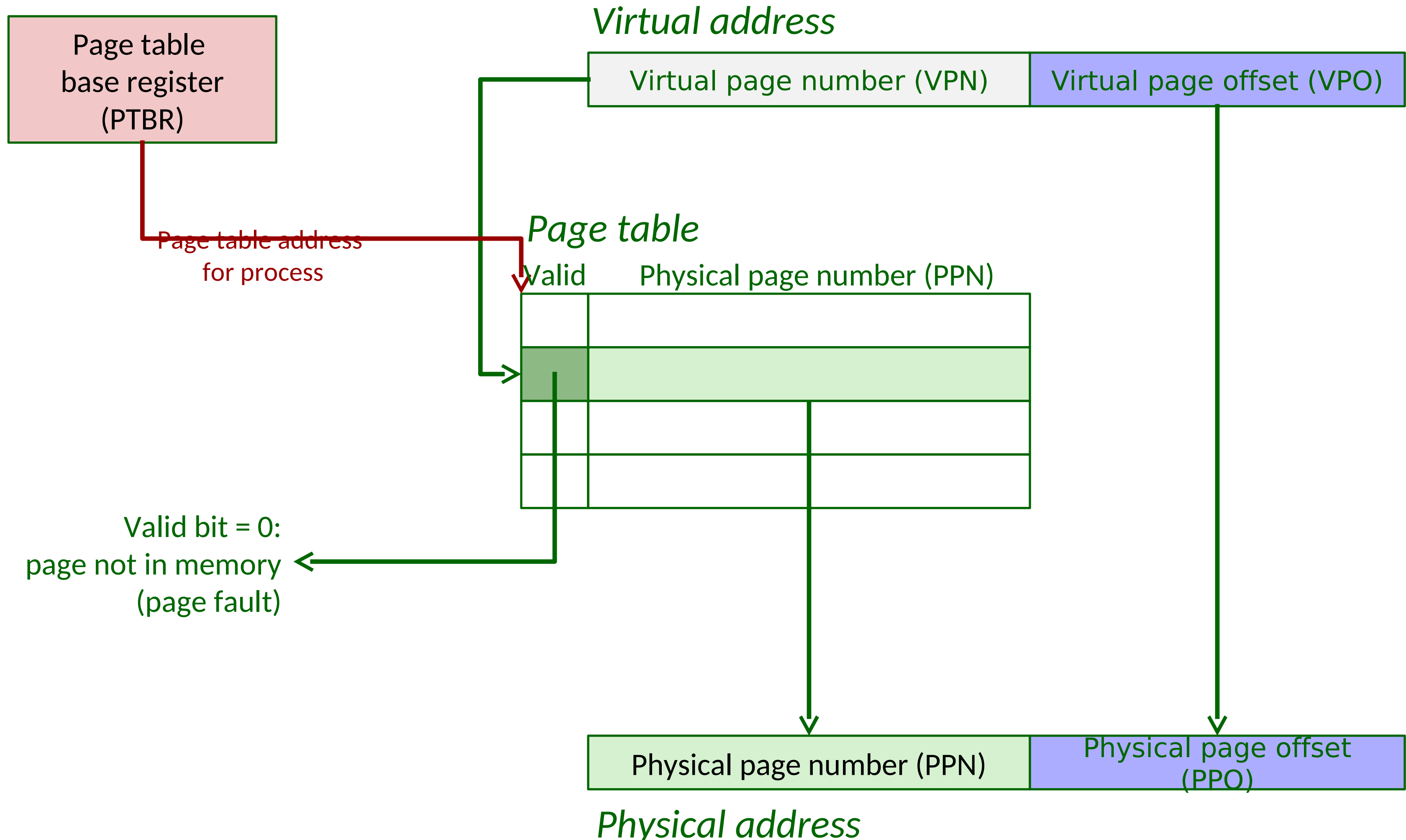
Address Translation: Page Tables

A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs

- **Per-process kernel data structure in DRAM**

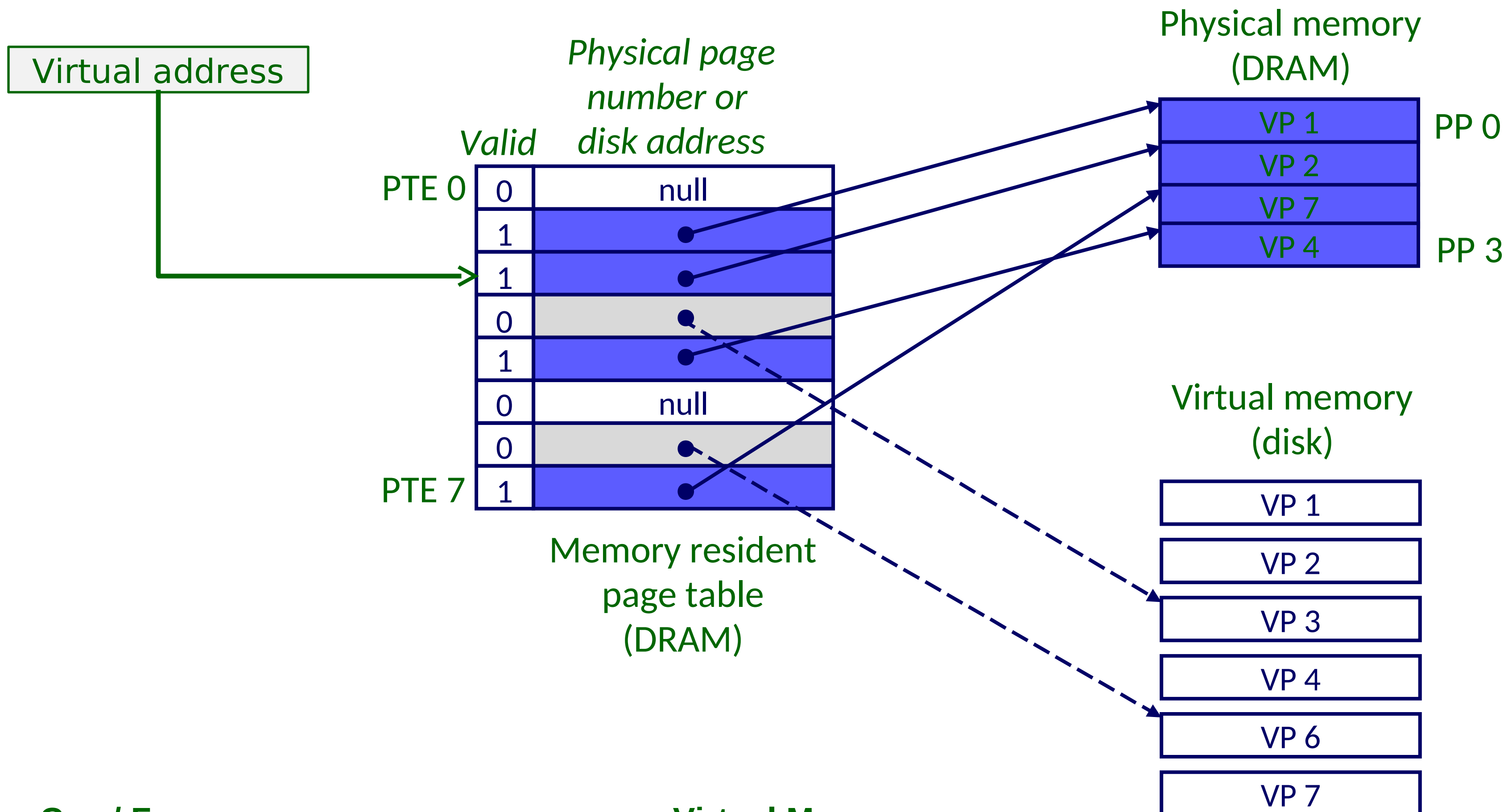


Address Translation With a Page Table



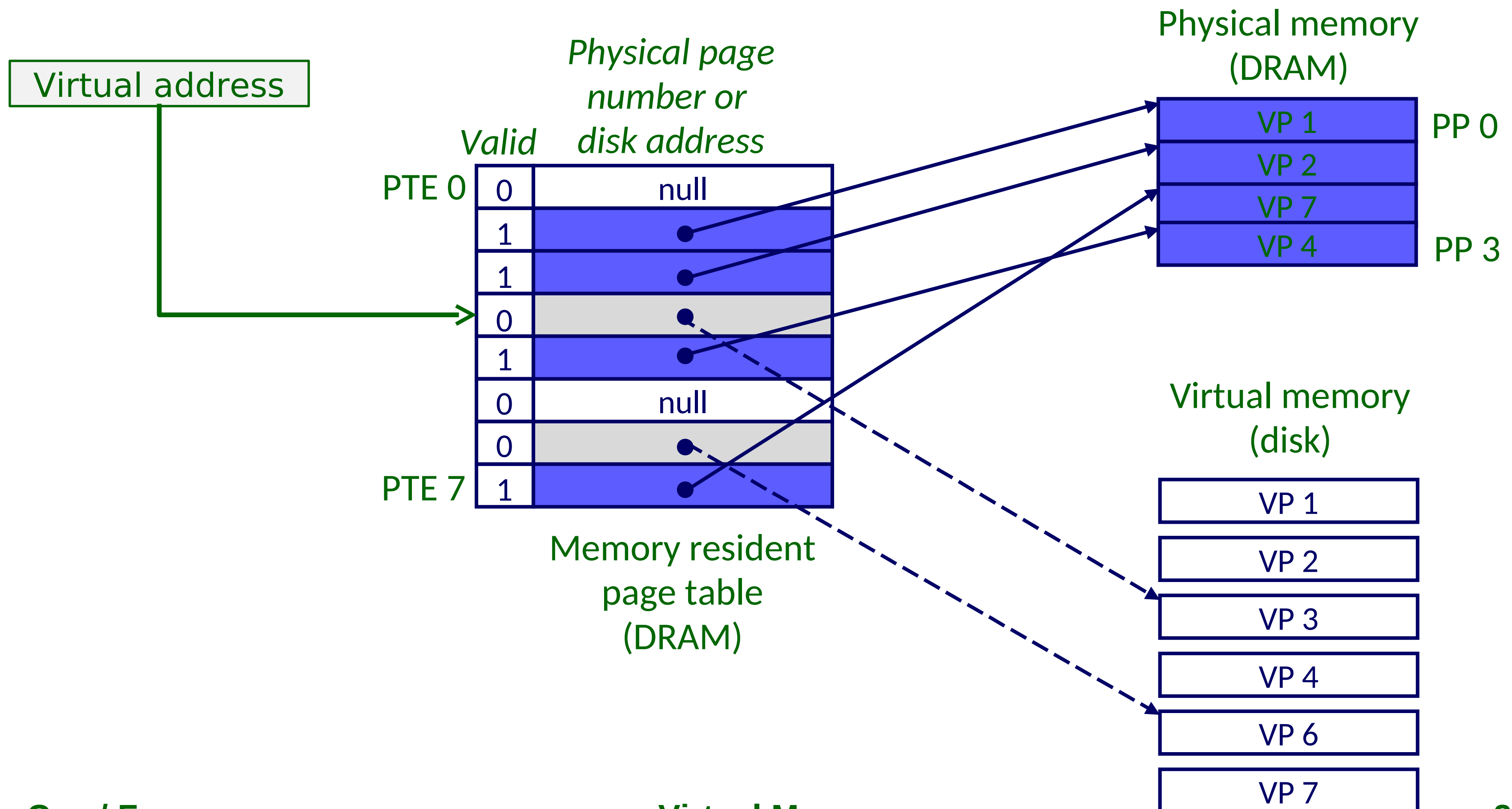
Page Hit

Page hit: **reference to VM word that is in physical memory**



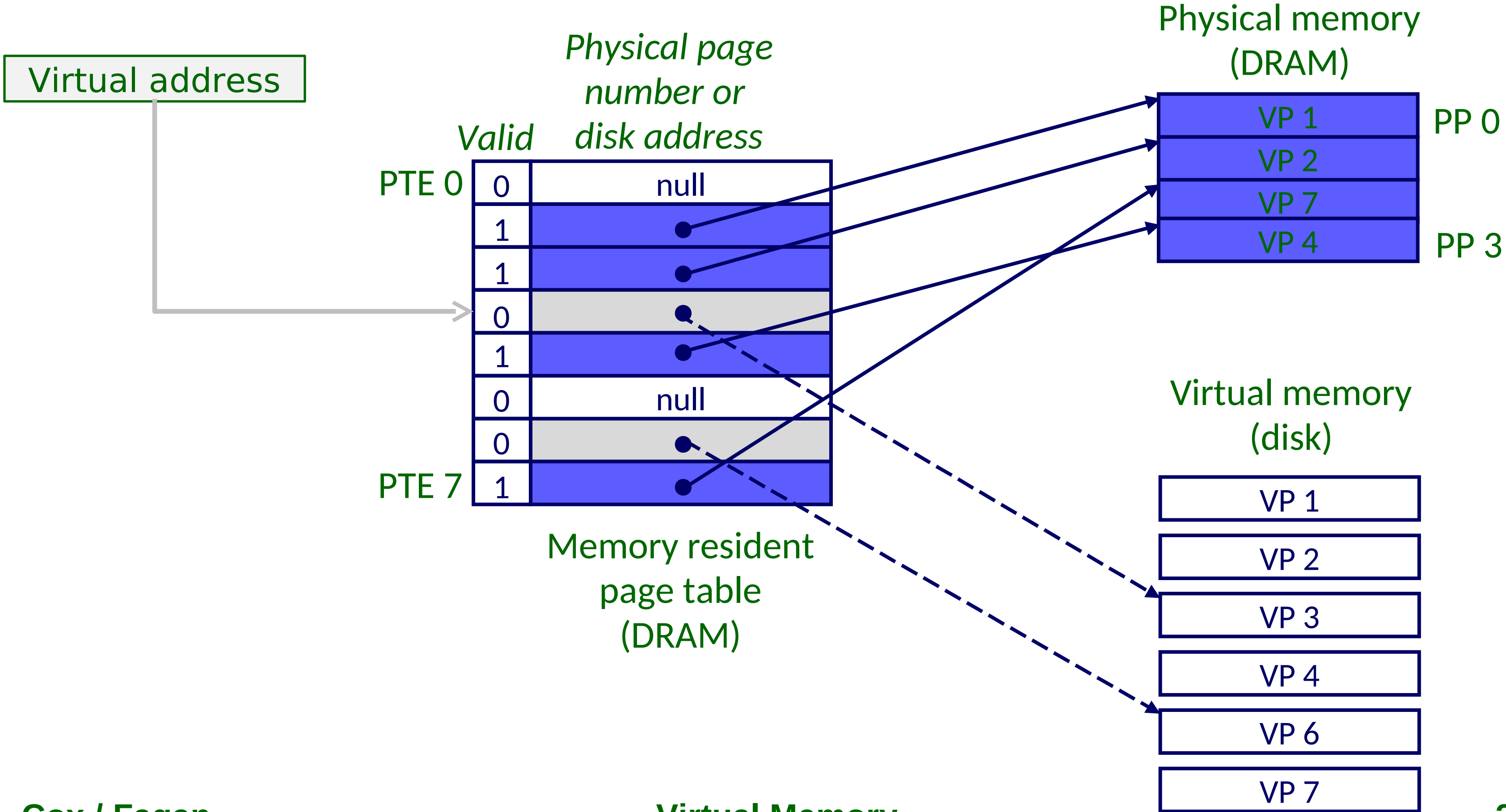
Page Miss

Page miss: reference to VM word that is not in physical memory



Handling Page Fault

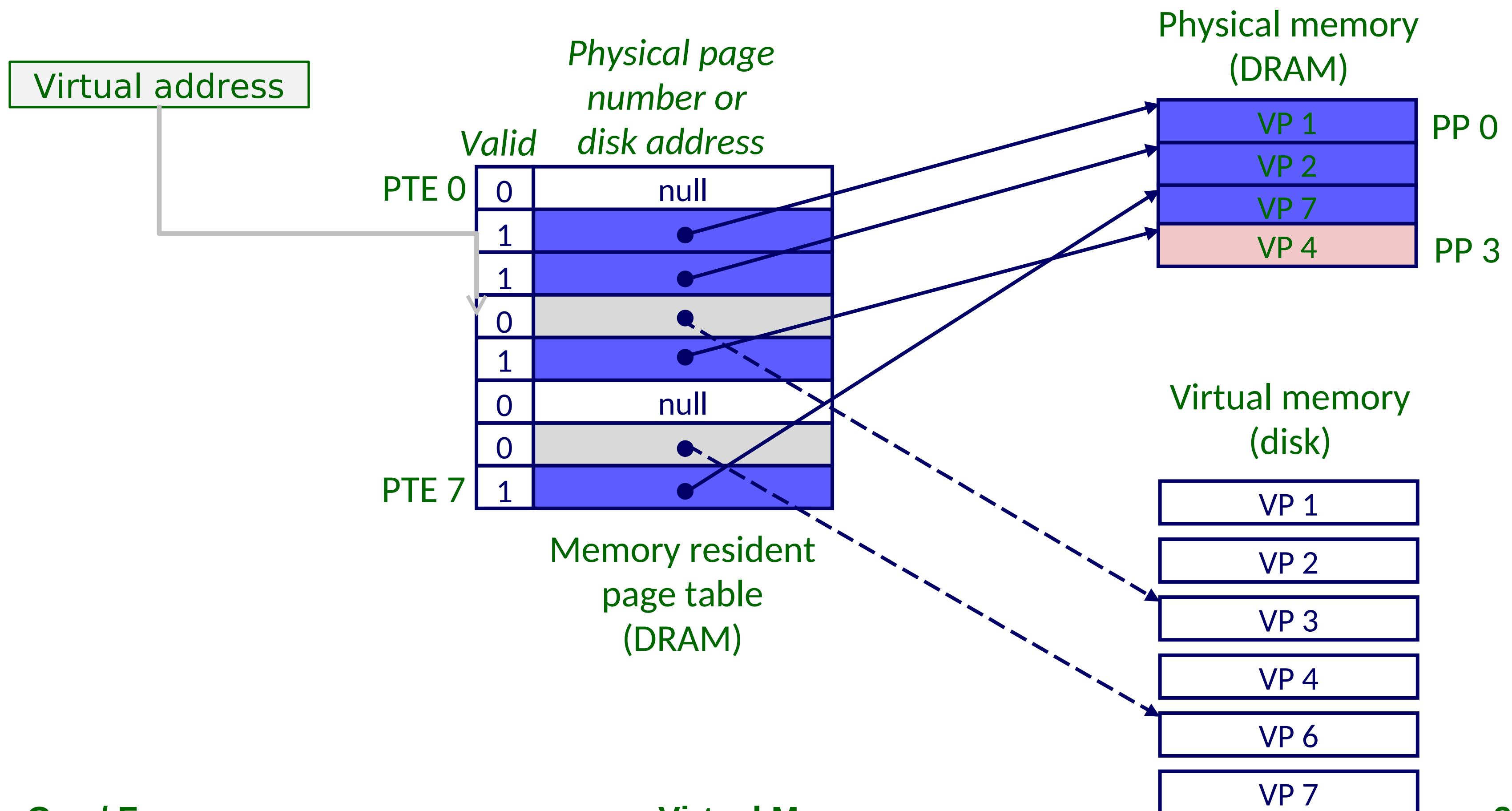
Page miss causes page fault (an exception)



Handling Page Fault

Page miss causes page fault (an exception)

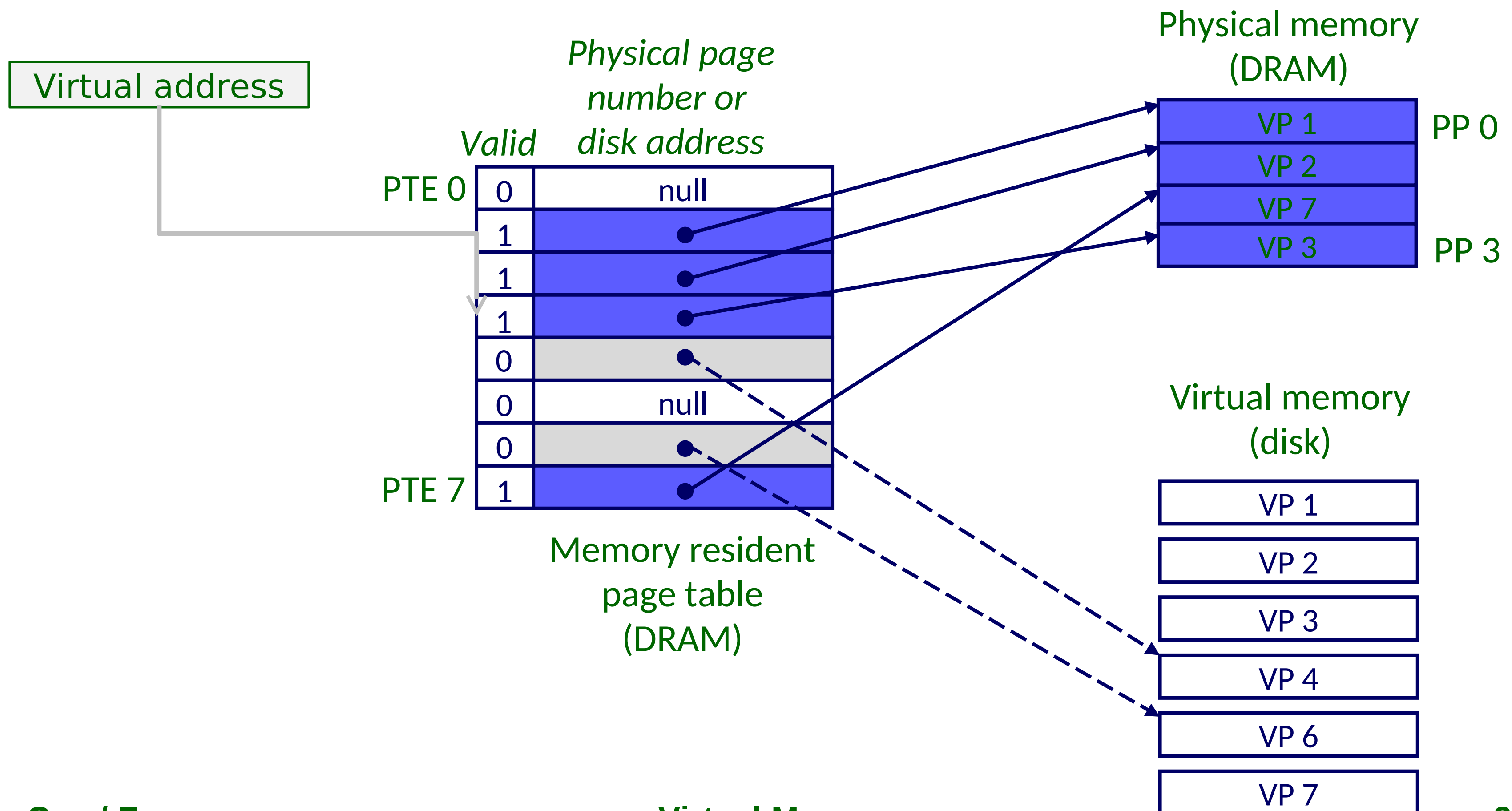
Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

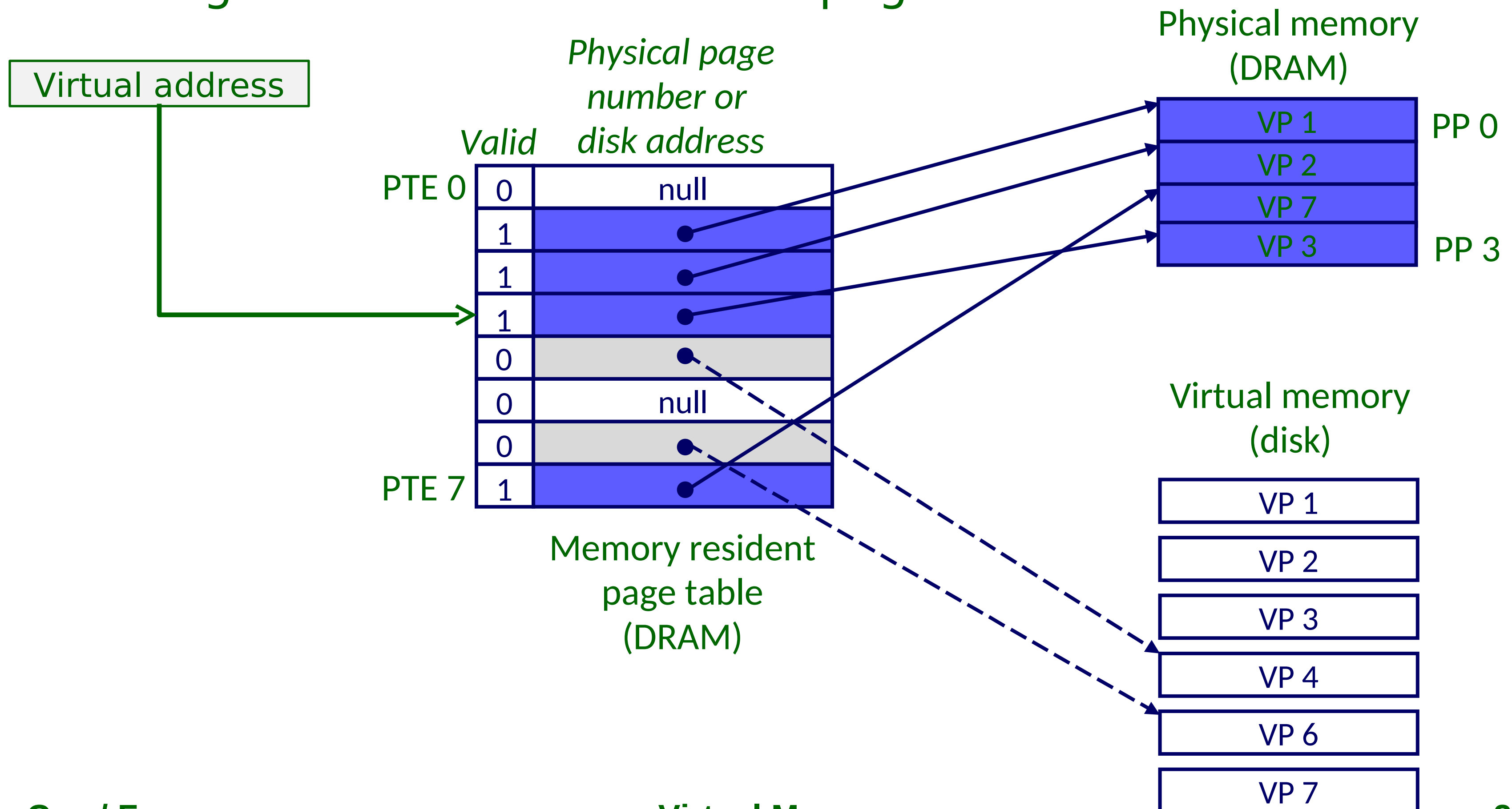
Page miss causes page fault (an exception)

Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Servicing a Page Fault

(1) Processor signals disk controller

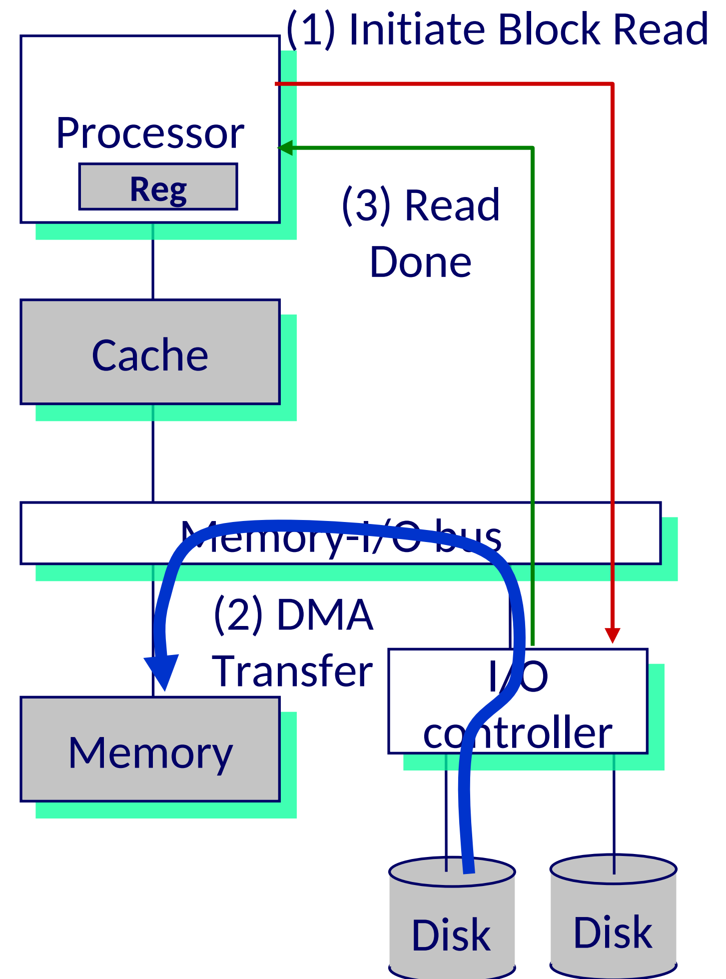
- **Read block of length P starting at disk address X and store starting at memory address Y**

(2) Read occurs

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

(3) Controller signals completion

- **Interrupts processor**
- **OS resumes suspended process**



Why does it work? Locality

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 for one process after initial compulsory misses**

If (SUM(working set sizes) > main memory size)

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

Outline

Virtual memory (VM)

- Overview and motivation
- VM as tool for caching
- **VM as tool for memory management**
- VM as tool for memory protection
- Address translation
- mmap(), fork(), exec()

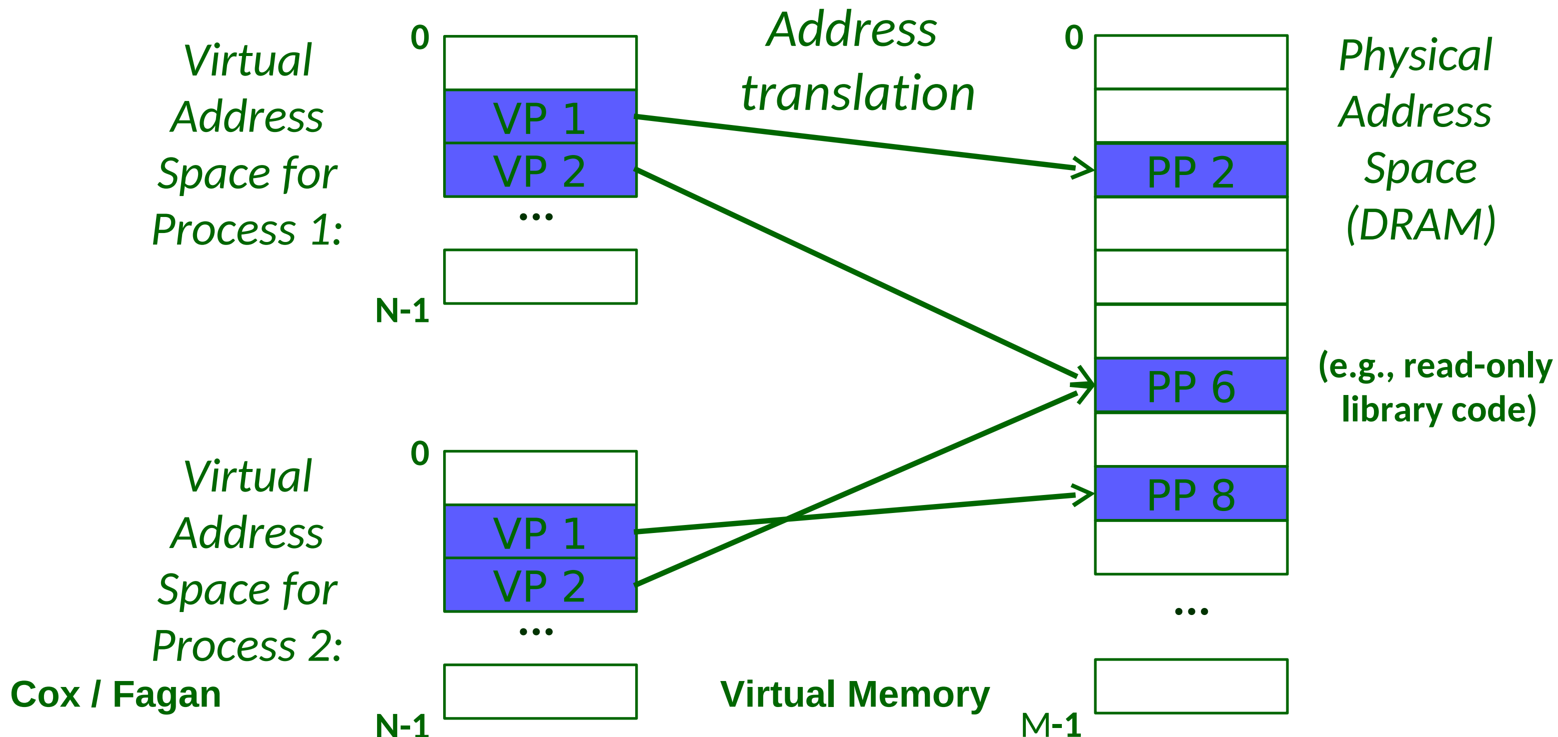
VM as a Tool for Memory Management

Memory allocation

- **Each virtual page can be mapped to any physical page**
- **A virtual page can be stored in different physical pages at different times**

Sharing code and data among processes

- **Map virtual pages to the same physical page (here: PP 6)**



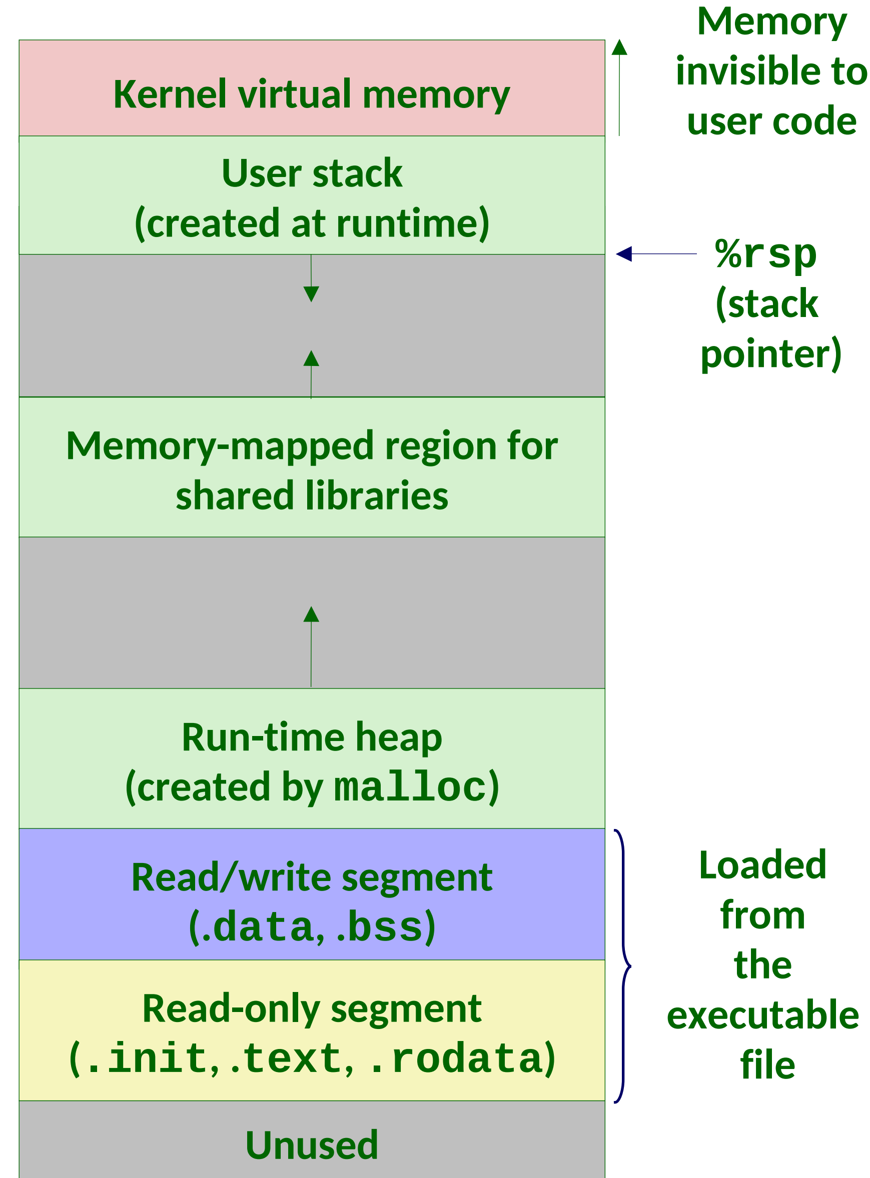
Simplifying Linking and Loading

Linking

- Each program has similar virtual address space
- Code, shared libraries, and stack can start at the same address in each process

Loading

- `execve()` allocates virtual pages for .text and .data sections
  = creates PTEs marked as invalid
- Pages within the `.text` and `.data` sections are loaded on demand by the virtual memory system



Outline

Virtual memory (VM)

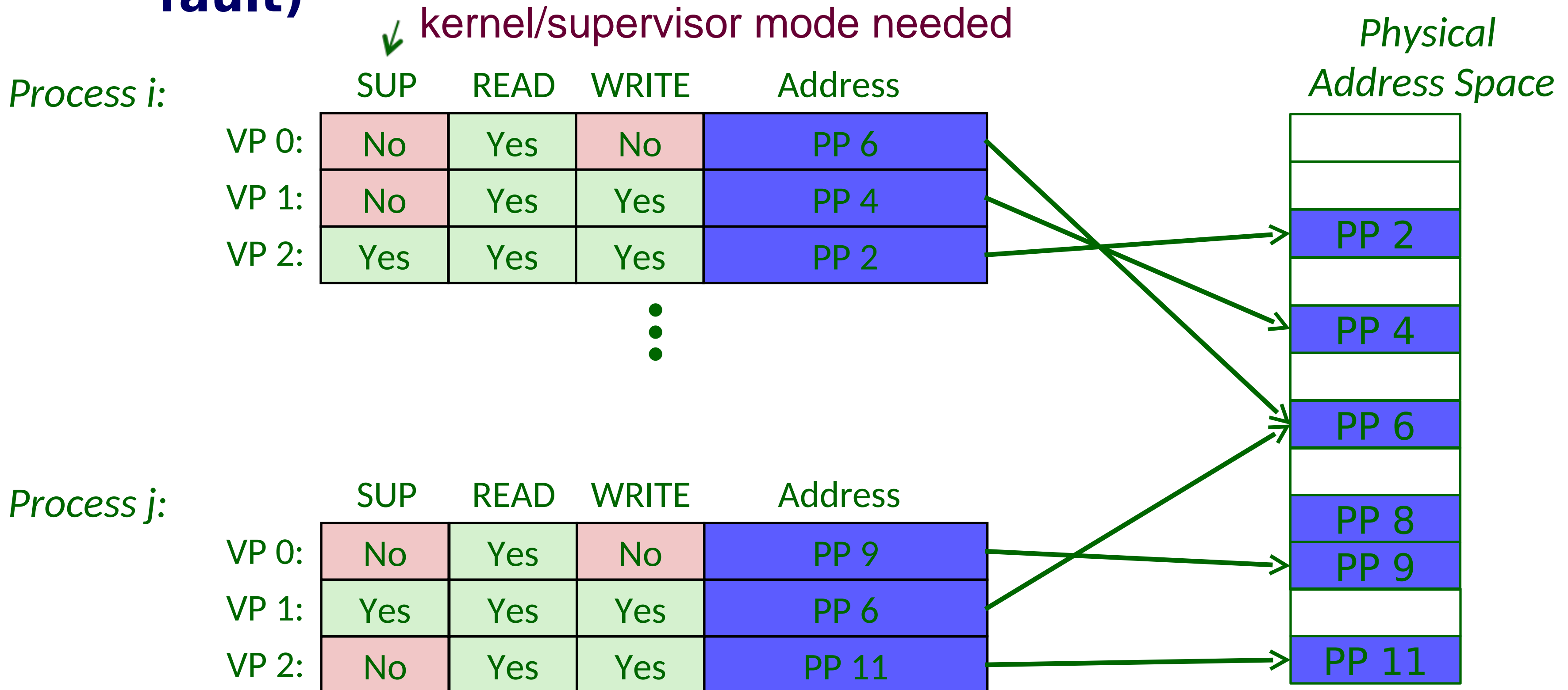
- Overview and motivation
- VM as tool for caching
- VM as tool for memory management
- **VM as tool for memory protection**
- Address translation
- mmap(), fork(), exec()

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)**

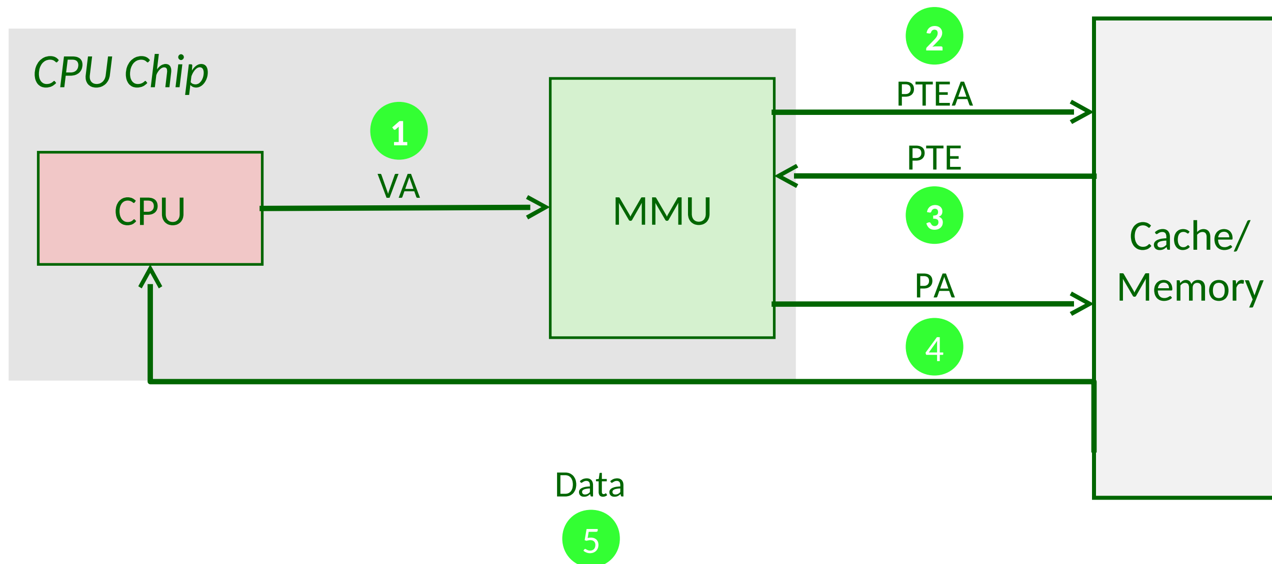


Outline

Virtual memory (VM)

- Overview and motivation
- VM as tool for caching
- VM as tool for memory management
- VM as tool for memory protection
- **Address translation**
- mmap(), fork(), exec()

Address Translation: Page Hit



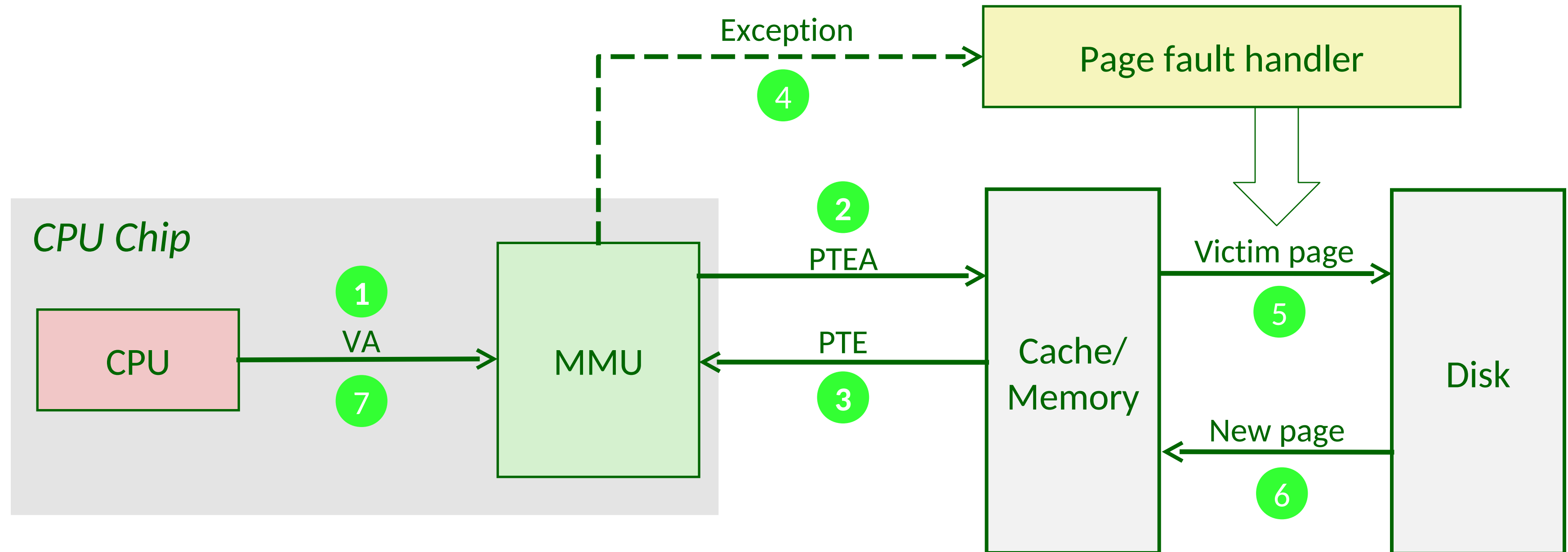
1) Processor sends virtual address to MMU

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

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU

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

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

5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Page Tables are often Multi-level in reality; accessing them is slow

Given:

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

Problem:

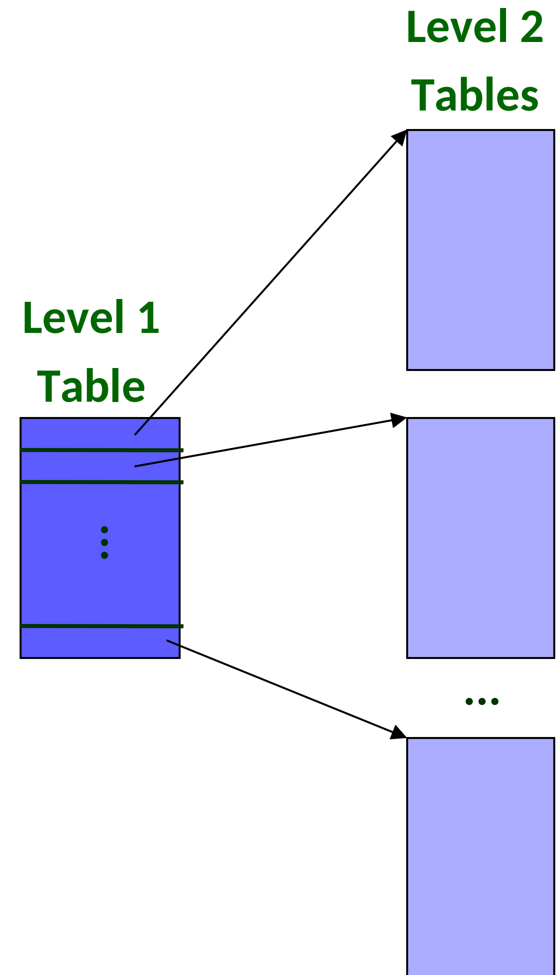
- Would need a 512 GB page table!
  - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes

Hardware solution:

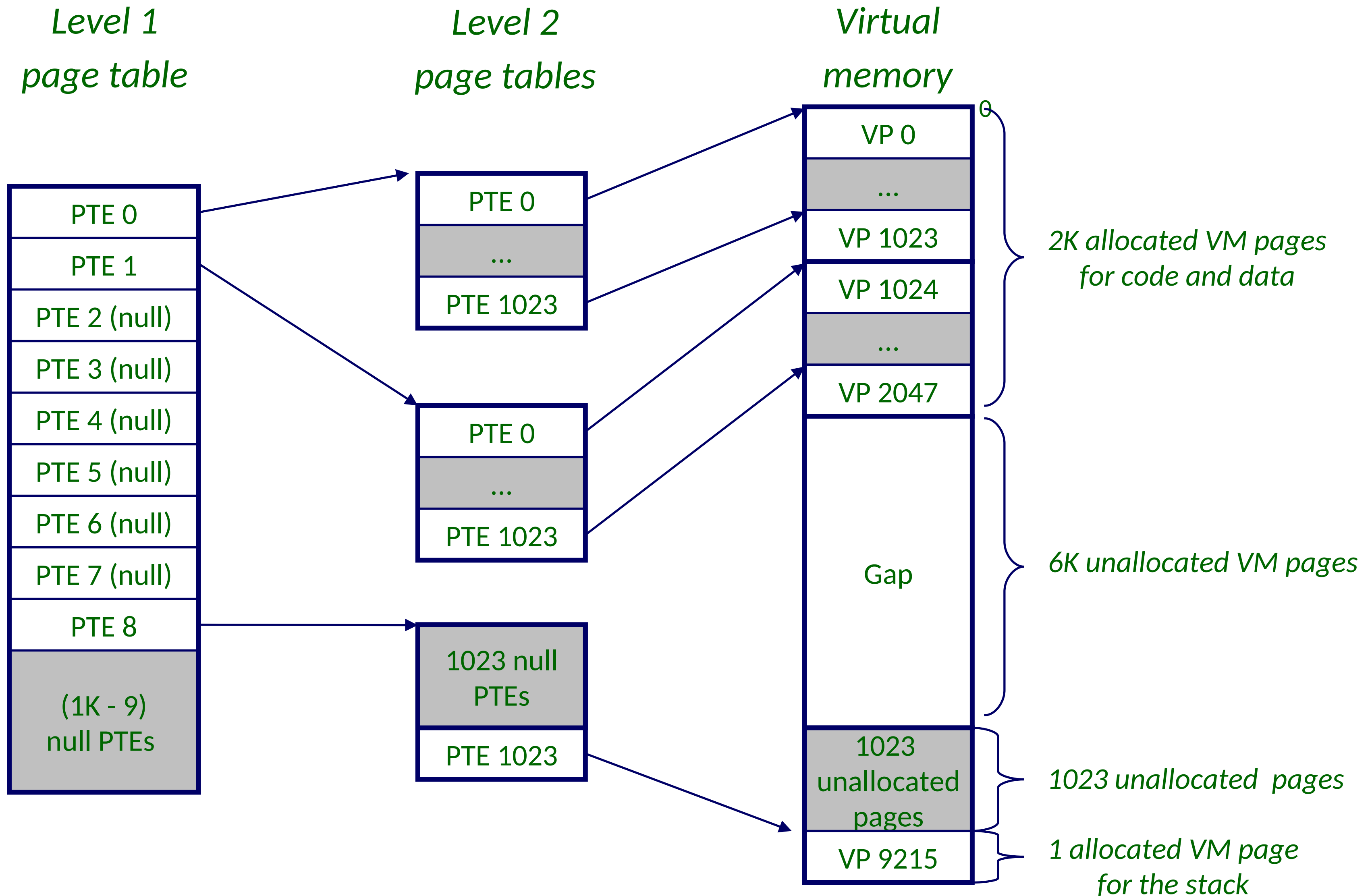
- Multi-level page tables
- Example: 2-level page table
  - Level 1 table: each PTE points to a level 2 page table
  - Level 2 table: each PTE points to a physical page

Complementary software:

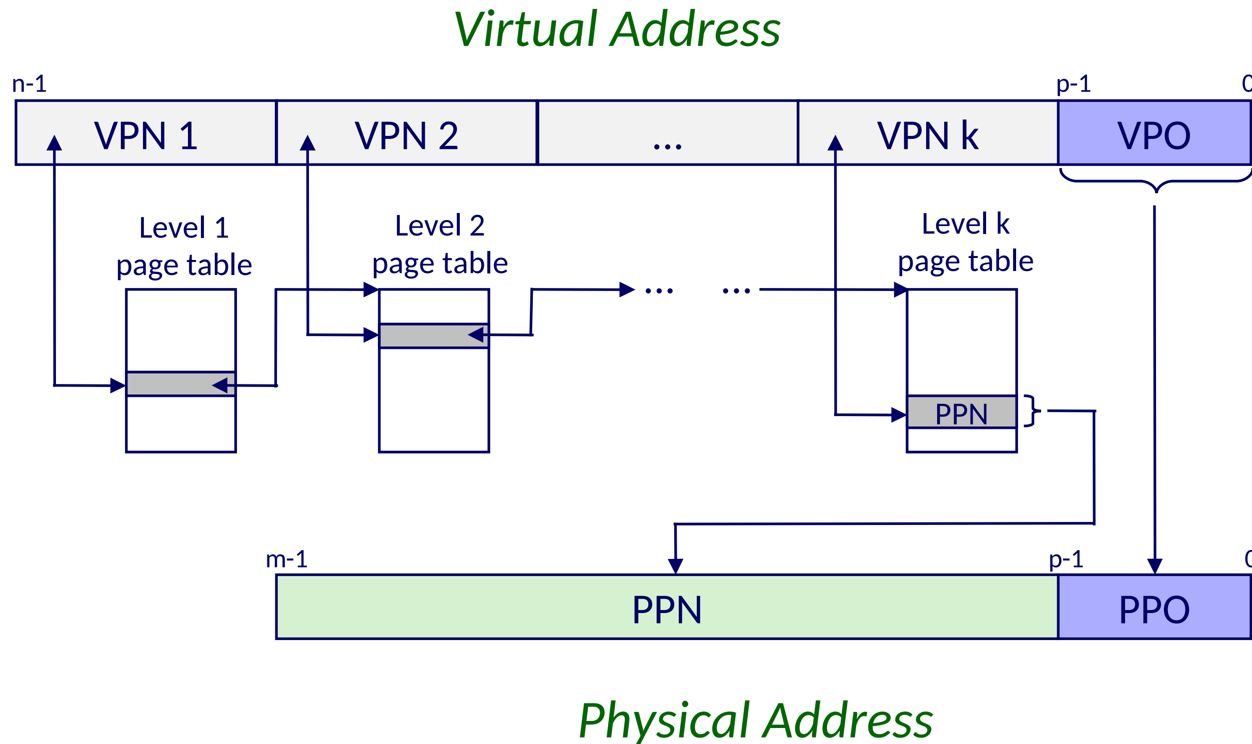
- Level 1 table stays in memory
- Level 2 tables paged in and out like other data



A Two-Level Page Table Hierarchy



Translating with a k-level Page Table



Speeding up Translation with a TLB

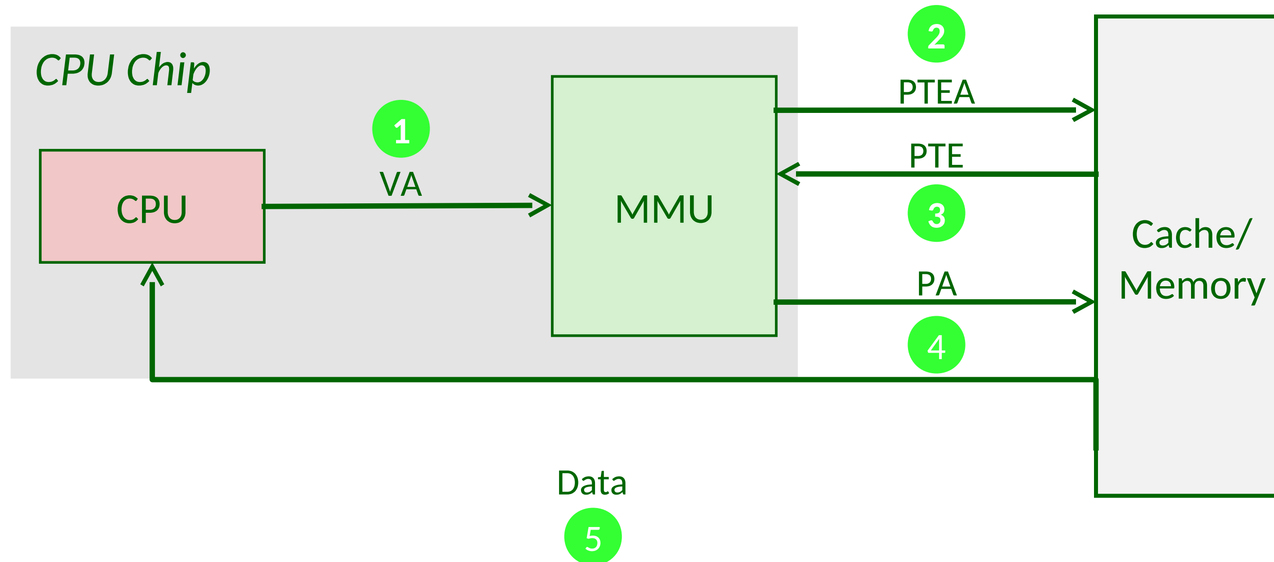
Page table entries (PTEs) are cached in L1 like any other memory word

- **PTEs may be evicted by other data references**
- **PTE hit still incurs memory access delay**

Solution: *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**

Without TLB



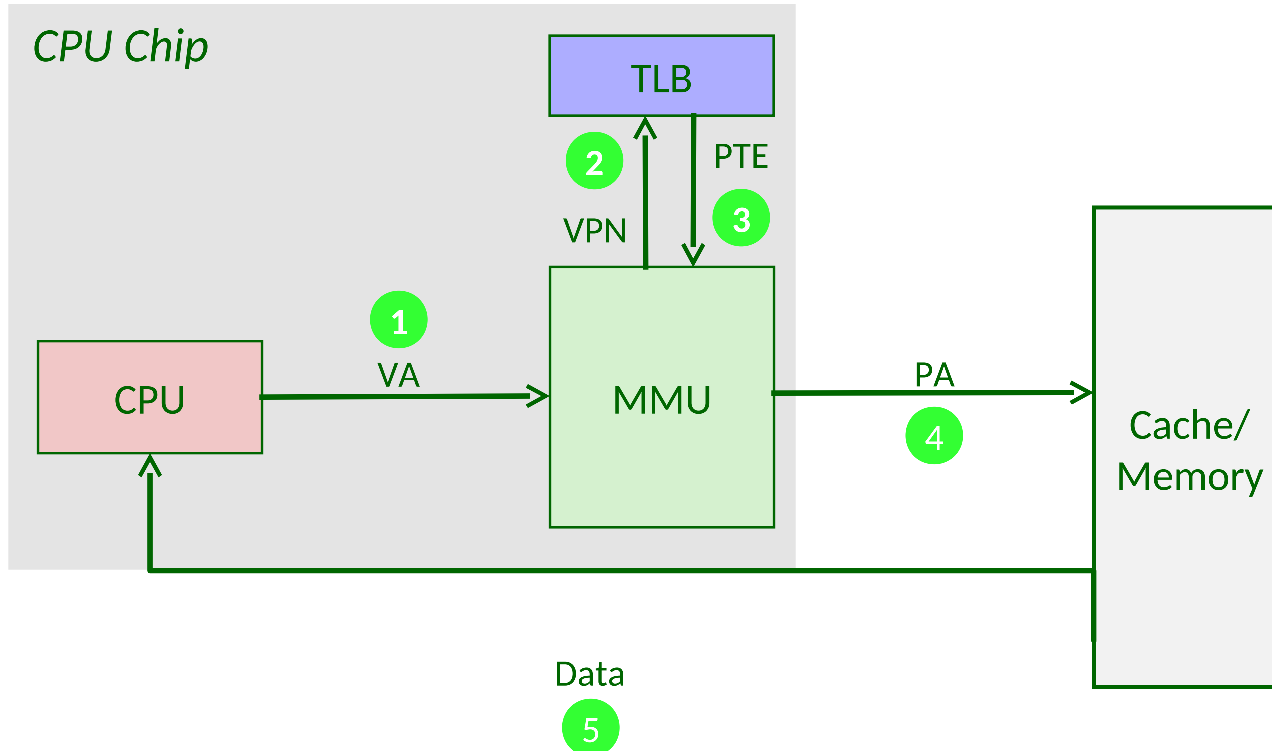
1) Processor sends virtual address to MMU

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

4) MMU sends physical address to cache/memory

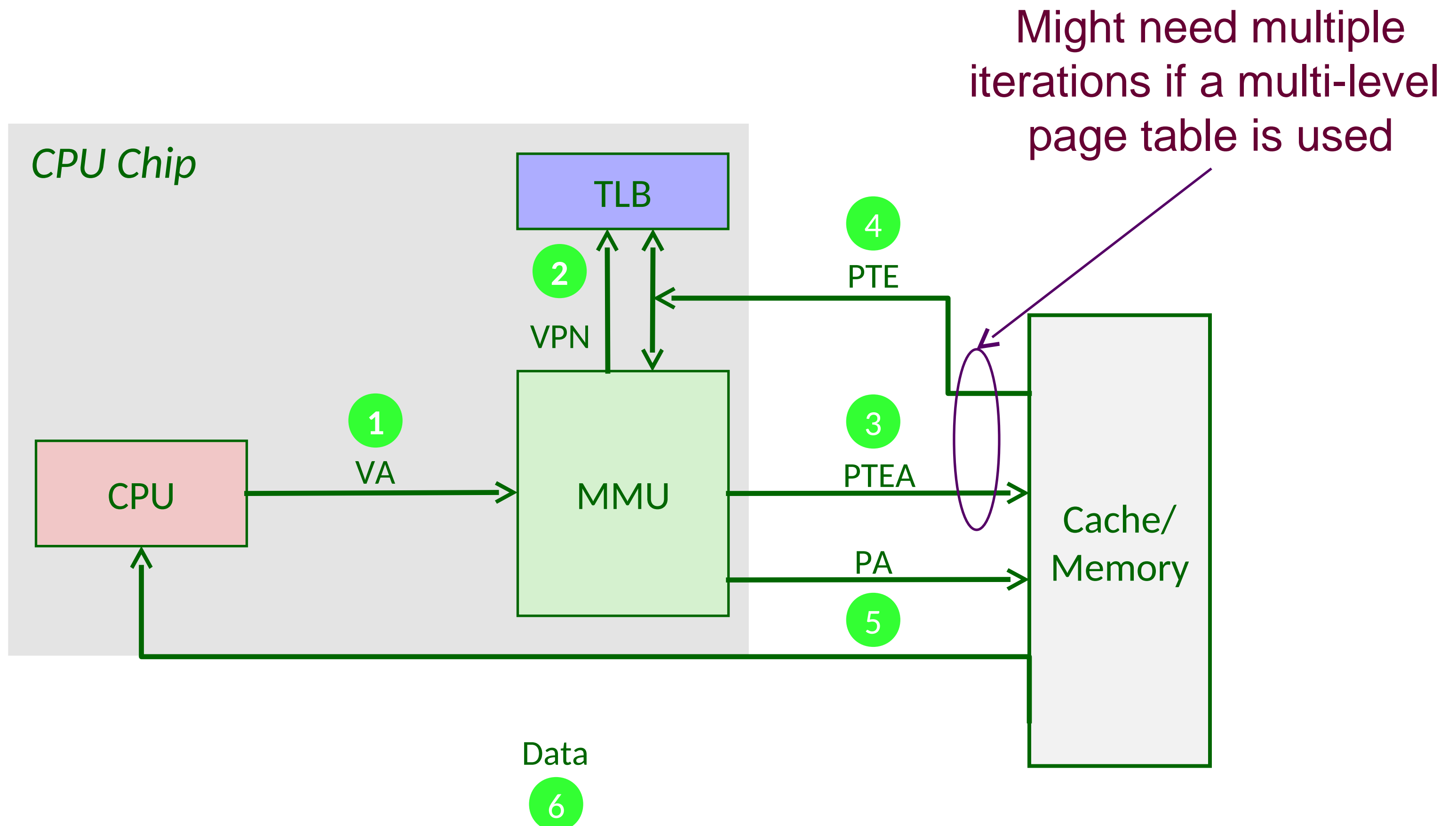
5) Cache/memory sends data word to processor

TLB Hit



A TLB hit eliminates a memory access

TLB Miss



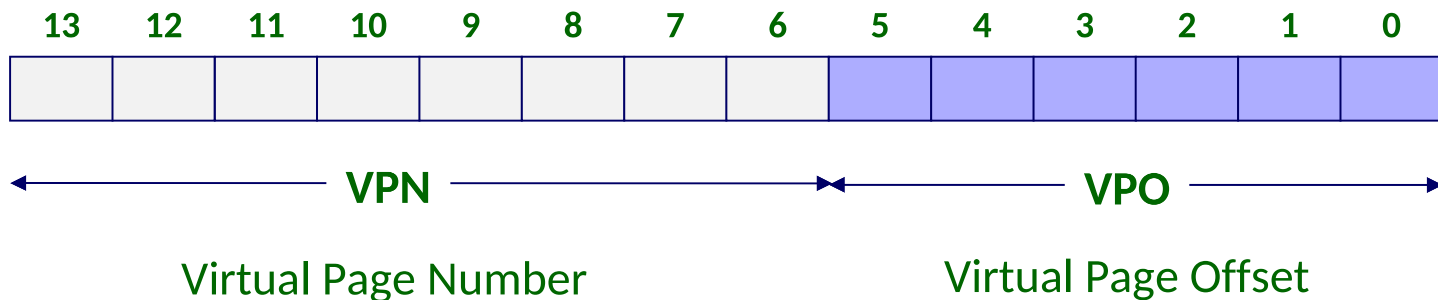
A TLB miss incurs an add'l memory access (the PTE)

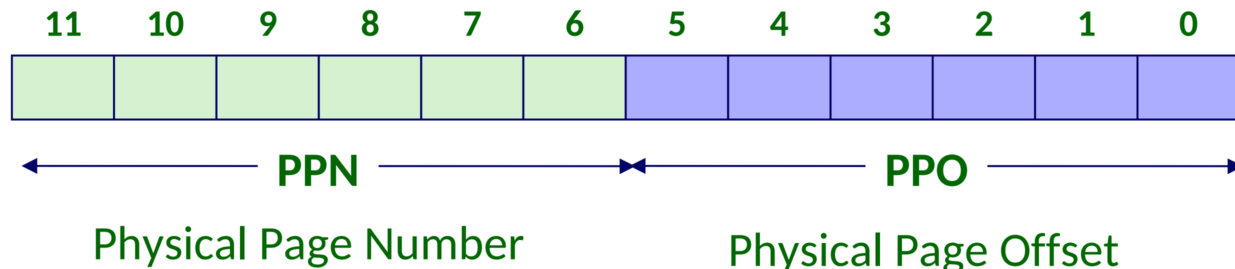
Fortunately, TLB misses are rare

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)

Assume all other entries are invalid

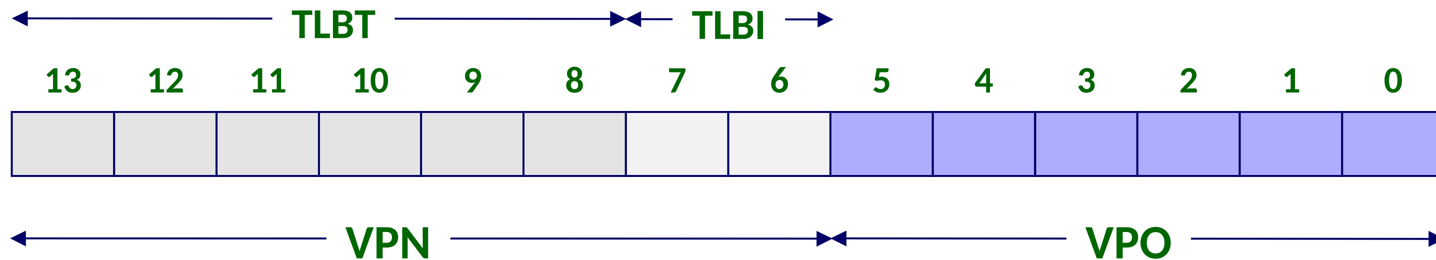
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

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

Simple Memory System TLB

16 entries

4-way associative



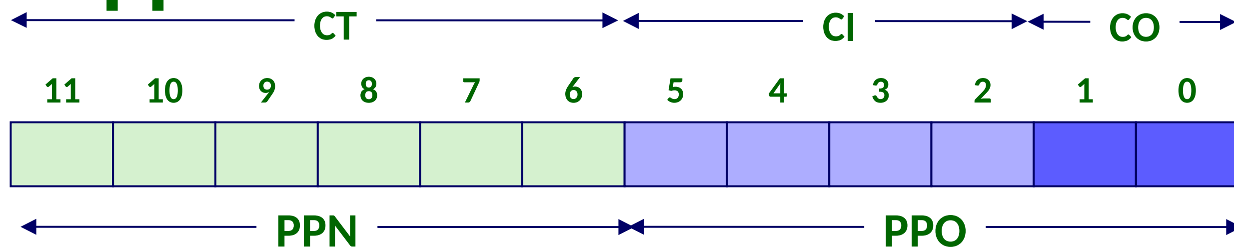
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System Cache

16 lines, 4-byte block size

Physically addressed

Direct mapped

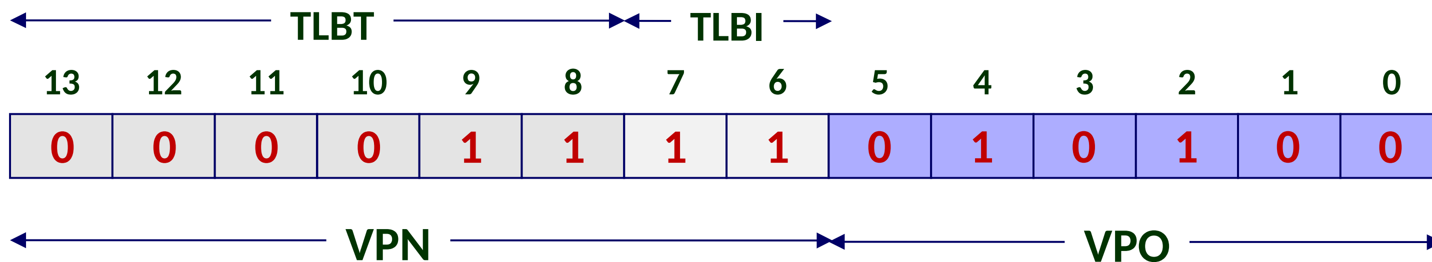


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

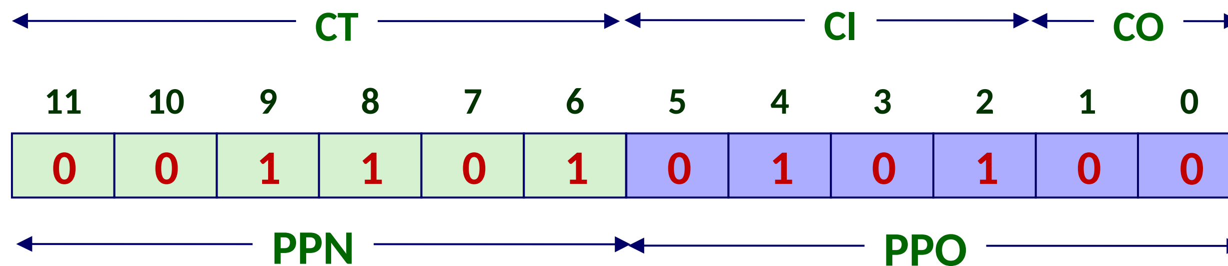
Address Translation Example #1

Virtual Address: 0x03D4



VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

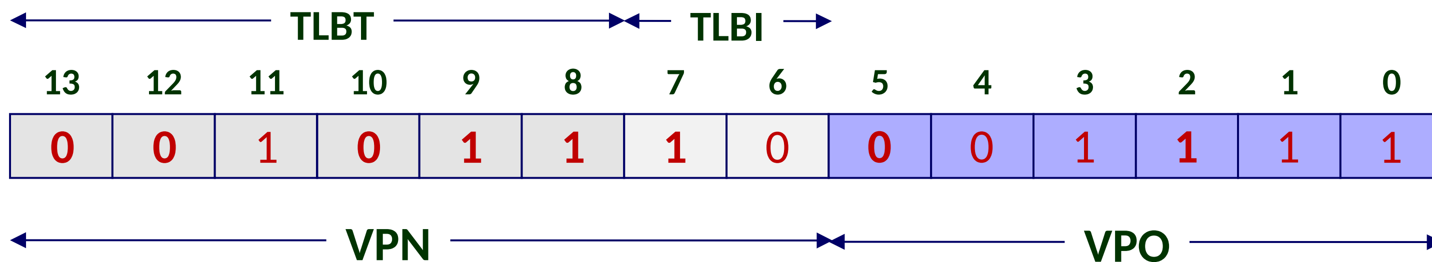
Physical Address



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

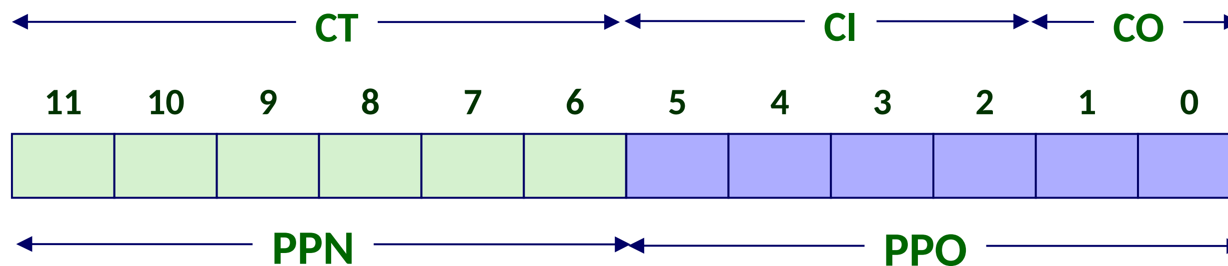
Address Translation Example #2

Virtual Address: 0x0B8F



VPN 0x2E **___ TLBI** 2 **___ TLBT** 0x0B **____ TLB Hit?** N **__ Page Fault?** Y **__ PPN: ____** TBD

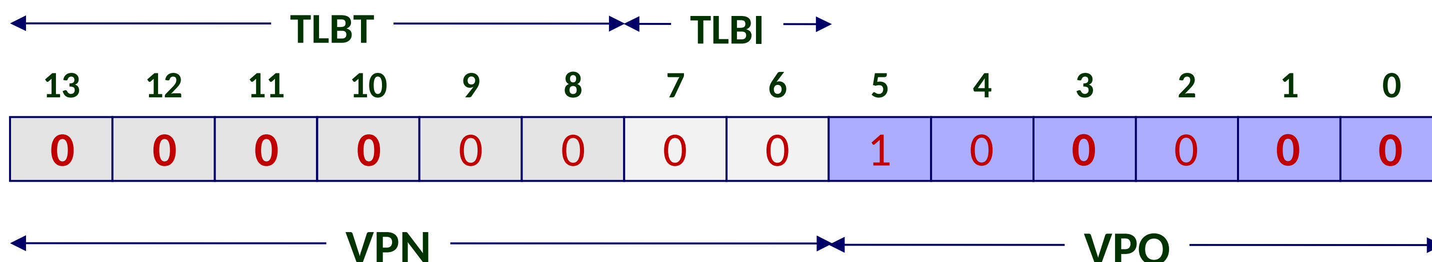
Physical Address



CO ___ CI___ CT ____ Hit? __ Byte: ____

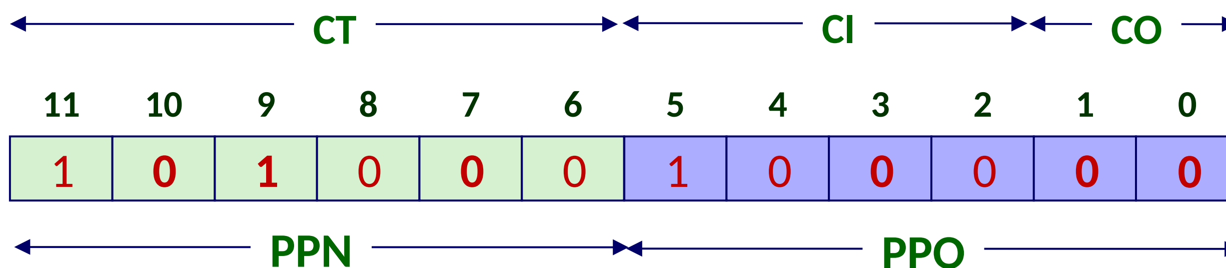
Address Translation Example #3

Virtual Address: `0x0020`



VPN 0x00 **TLBI** 0 **TLBT** 0x00 **TLB Hit?** N **Page Fault?** N **PPN:** 0x28

Physical Address



CO 0 **CI** 0x8 **CT** 0x28 **Hit?** N **Byte:** In DRAM, don't know value

Outline

Virtual memory (VM)

- Overview and motivation
- VM as tool for caching
- VM as tool for memory management
- VM as tool for memory protection
- Address translation
- **mmap(), fork(), exec()**

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 (demand-zero)**
  - **Once the page is written to (dirtied), it is like any other page**

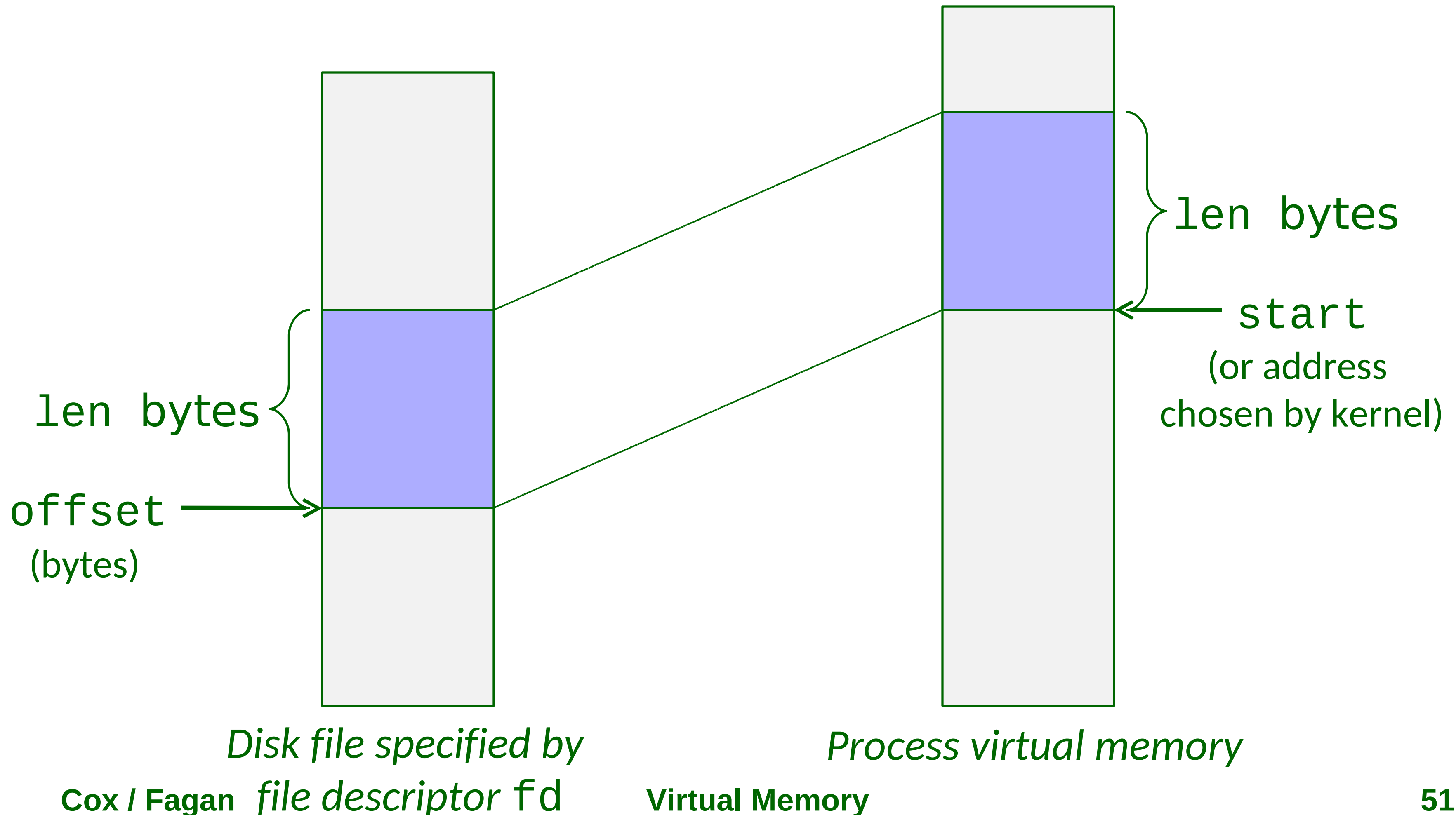
Dirty pages are swapped back and forth between a special swap area of the disk.

Key point: **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, size_t len,
           int prot, int flags, int fd, off_t offset)
```



User-Level Memory Mapping

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

Map `len` bytes starting at offset `offset` of the file specified by file descriptor `fd`, preferably at address `start`

- **`start`: may be 0 for "pick an address"**
- **`prot`: PROT_READ, PROT_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

```
/* mmap.c - a program that uses mmap to copy itself to stdout */
/* include <unistd.h>, <sys/mman.h>, <sys/stat.h>, and <fcntl.h> */
int
main(void)
{
  struct stat stat;
  int fd;
  char *bufp;

  /* Open the file & get its size. */
  fd = open("./mmap.c", O_RDONLY);
  fstat(fd, &stat);

  /* Map the file to a new VM area. */
  bufp = mmap(NULL, stat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

  /* Write the VM area to stdout. */
  write(STDOUT_FILENO, bufp, stat.st_size);

  return (0);
}
```

`fork()` Revisited

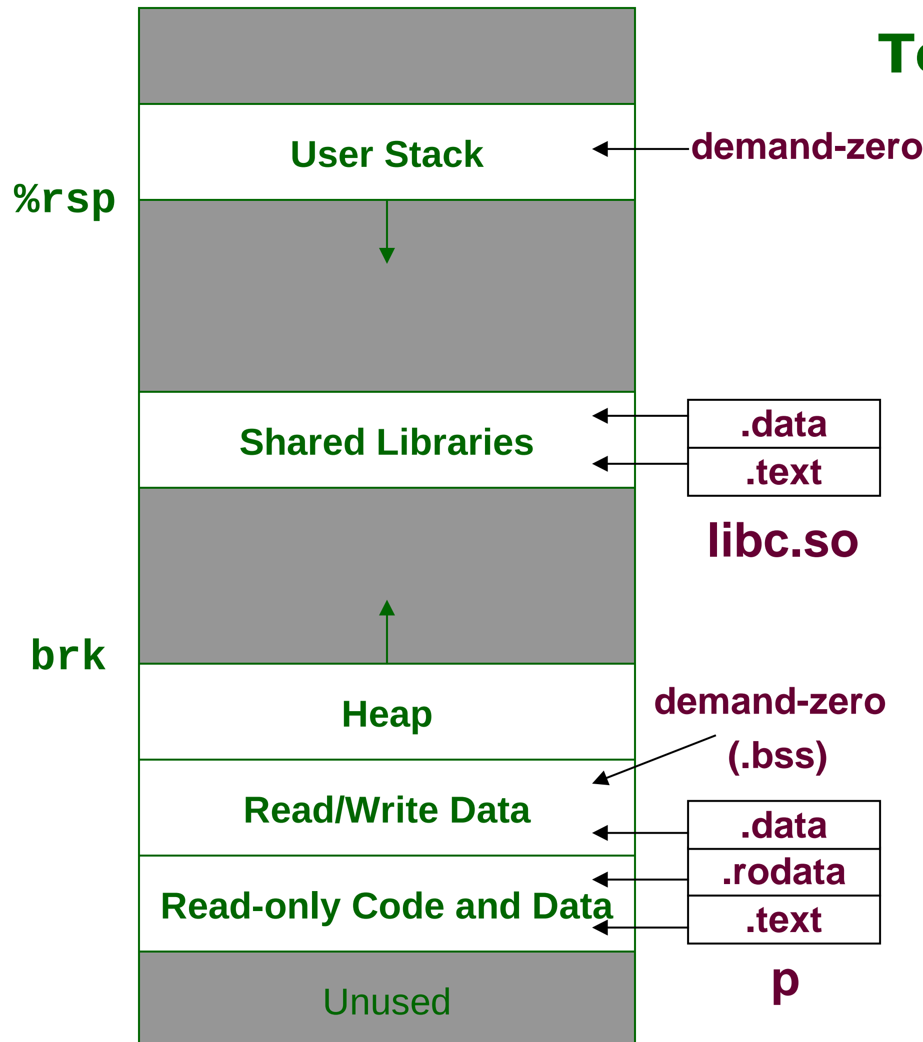
To create a new process using fork():

- **Make copies of the old process' page table, etc.**
  - The two processes now share all of their pages
- **Copy-on-write**
  - Allows each process to have a separate address space without copying all of the virtual pages
  - Make pages of writeable areas read-only
  - Flag these areas as private "copy-on-write" in OS
  - Writes to these pages will cause protection faults
    - Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions

Net result:

- **Processes have identical address spaces**
- **Copies are deferred until absolutely necessary**

exec() Revisited



To load p using exec:

- Delete existing page tables, etc.
- Create new page tables, etc.:
  - Stack/heap/.bss are anonymous, demand-zero
  - Code and data is mapped to ELF executable file p
- Shared libraries are dynamically linked and mapped
- Set program counter to entry point in .text
  - OS will swap in pages from disk as they are used

Virtual Memory

Supports many OS-related functions

- Process creation
  - Initial
  - Forking children
- Task switching
- Protection/sharing

Combination of hardware & software implementation

- Software manages page tables, page allocations
- Hardware reads page tables
  - Page fault when no entry
- Hardware enforcement of protection
  - Protection fault when invalid access

Next Time

System-Level I/O