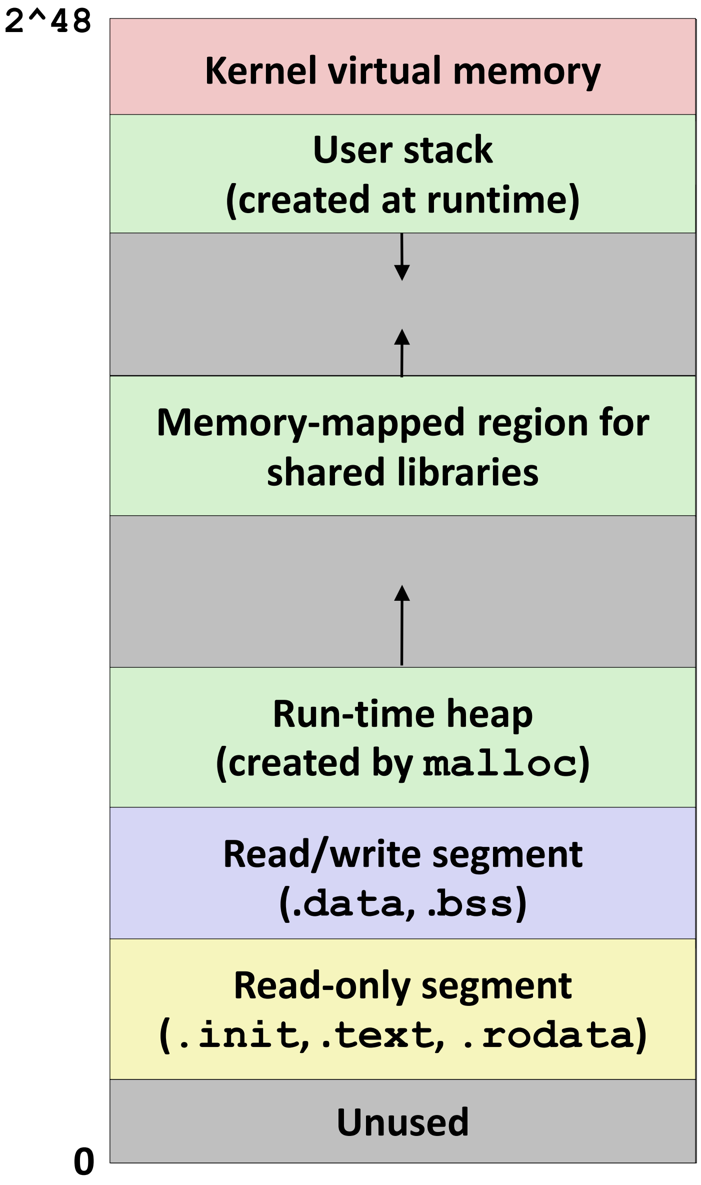
Virtual Memory:

Outline

- Why Virtual Memory
- How do we logically support it?
- How do we actually support it?
- Example!
- Summary

Virtual Memory

- **Consists of:**
  - Large linear address space
  - Independent of physical memory
  - Unique to each process
- **Alternative:**
  - Each process gets a portion of the physical DRAM : (



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Why Virtual Memory (VM)?

- **Uses main memory efficiently**
  - Use DRAM as a cache for parts of a virtual address space

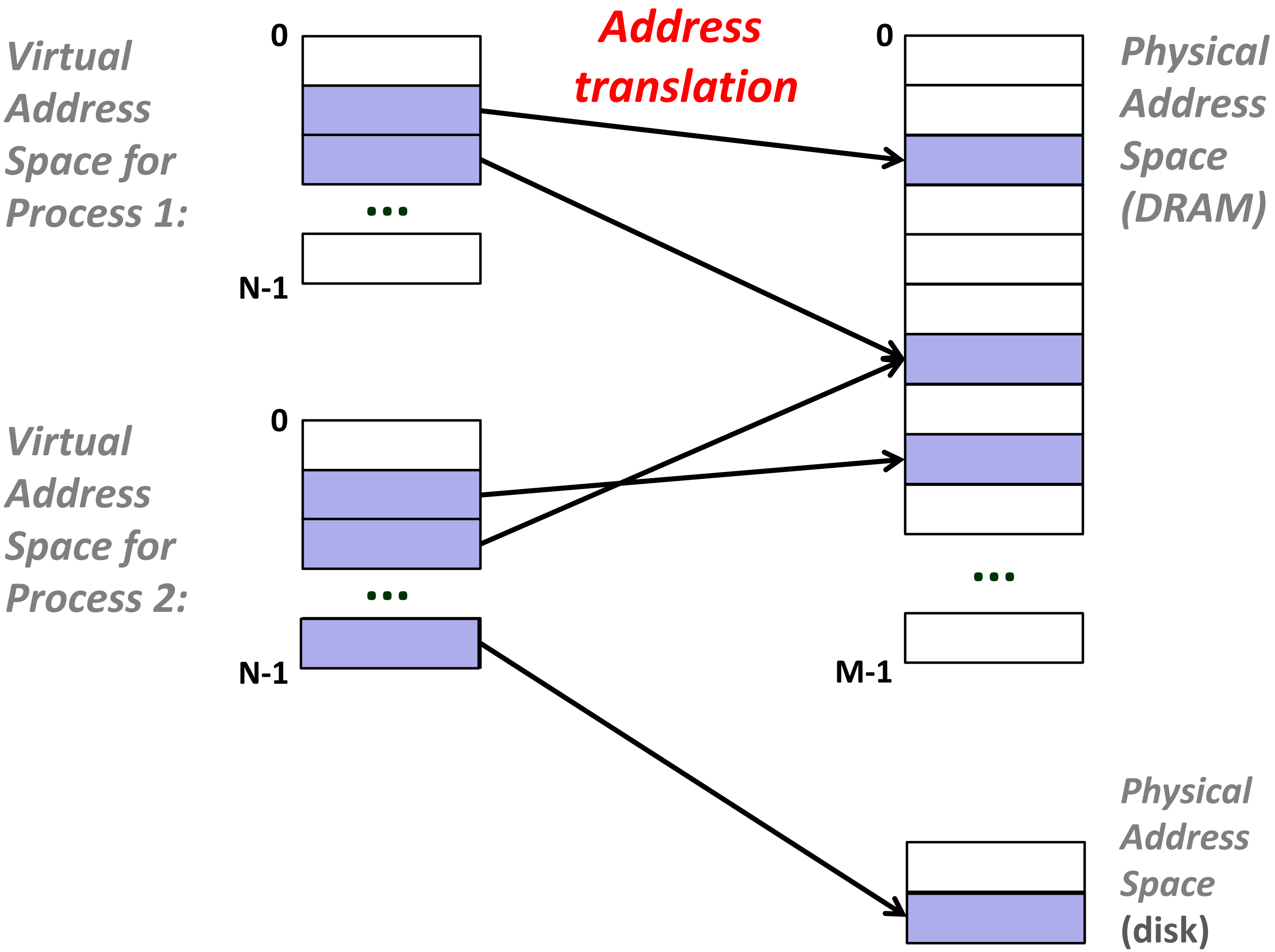
- **Simplifies memory management**
  - Each process gets the same uniform linear address space

- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code

So what’s the big deal?

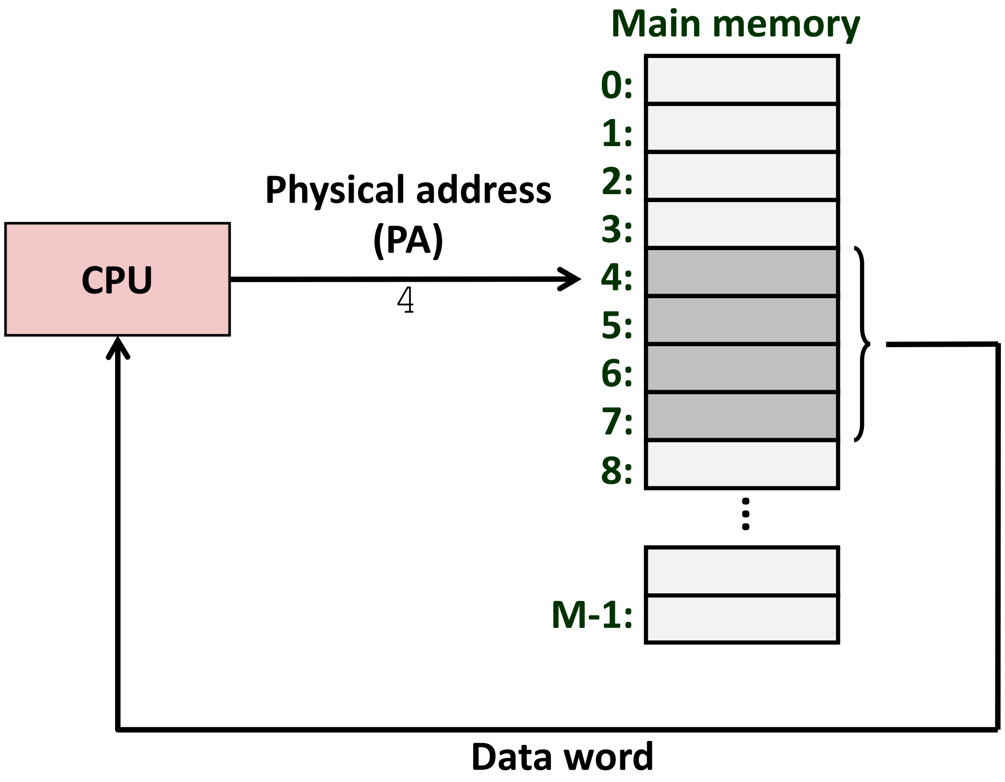
Virtual Memory >> Physical Memory

How to support Virtual Memory?


Virtual Address Space for Process 1:
0
•••
N-1
Address translation
0
Physical Address Space (DRAM)
•••
M-1
Virtual Address Space for Process 2:
0
•••
N-1
Physical Address Space
(disk)

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

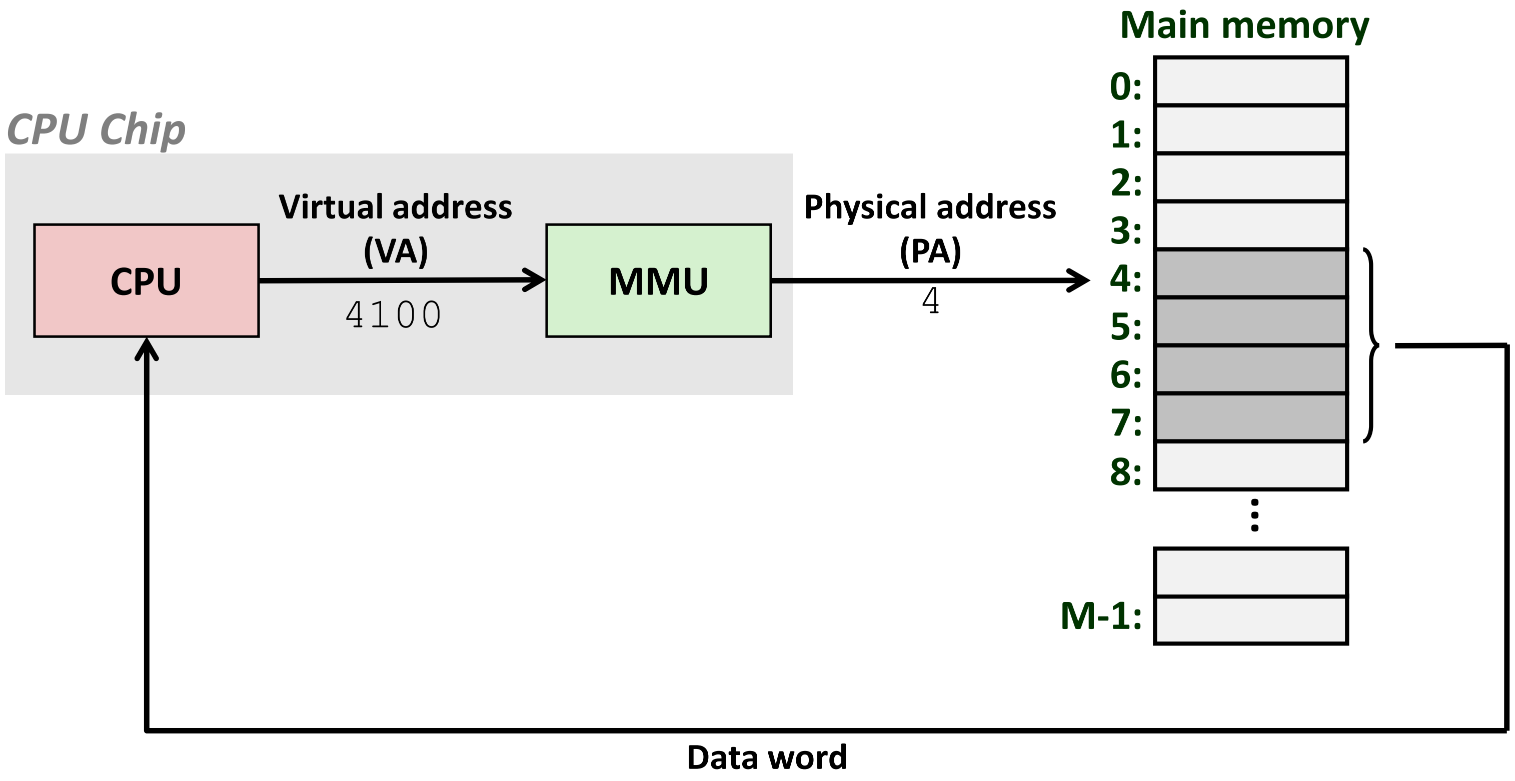
A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames (but not always…)

Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

A System Using Virtual Addressing



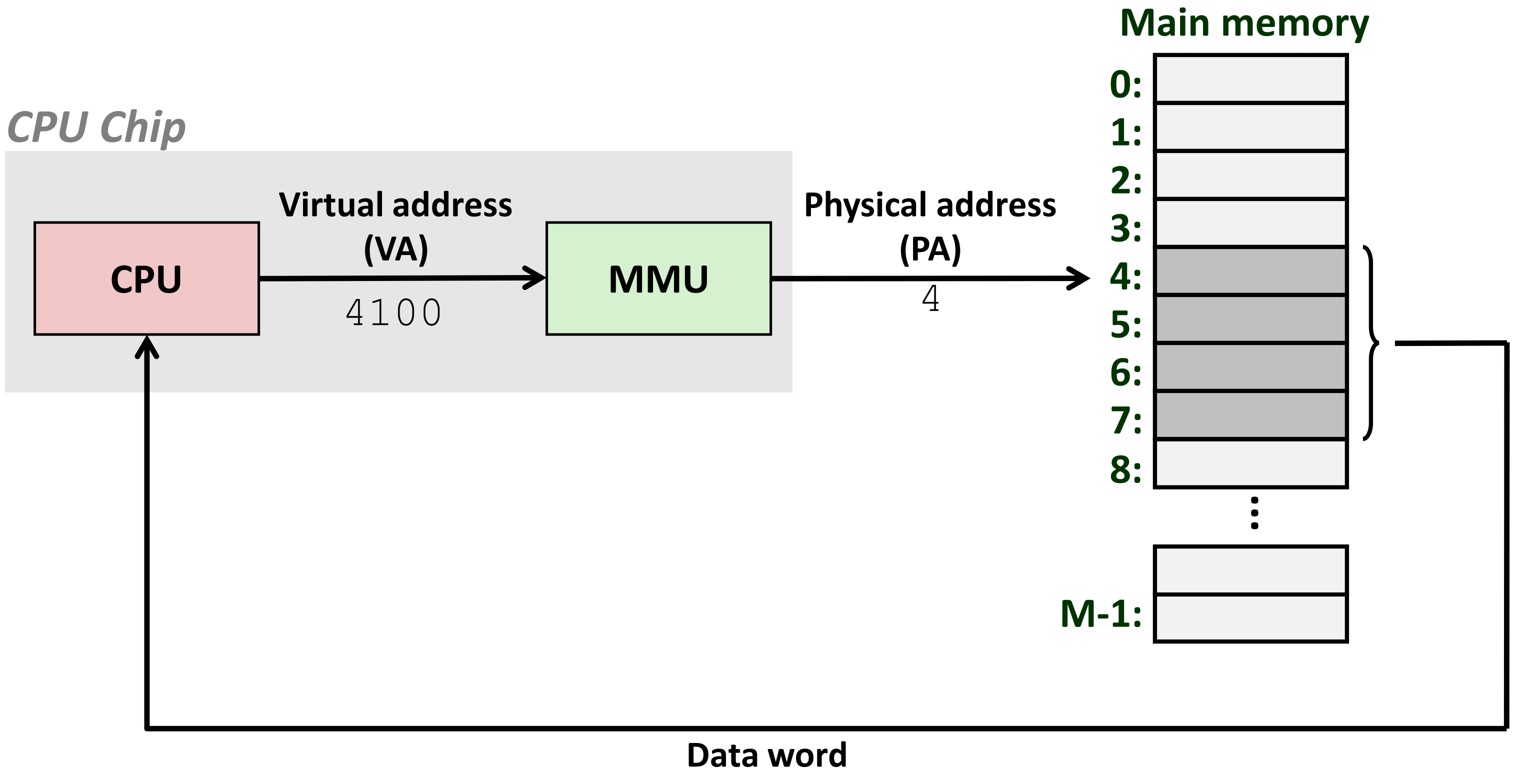
- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Basic Strategy

- **Paging**
  - Logically divide memory into chunks called “Pages”
  - Make Decision of where to put memory on Page Granularity
  - Size: typically 4 KB, sometimes 4 MB
- **Flexible Translation**
  - Any virtual page (VP) can be mapped to any physical page (PP)
    - Fully associative
    - Requires a “large” mapping function – different from cache memories
    - Too complicated and open-ended to be implemented in hardware

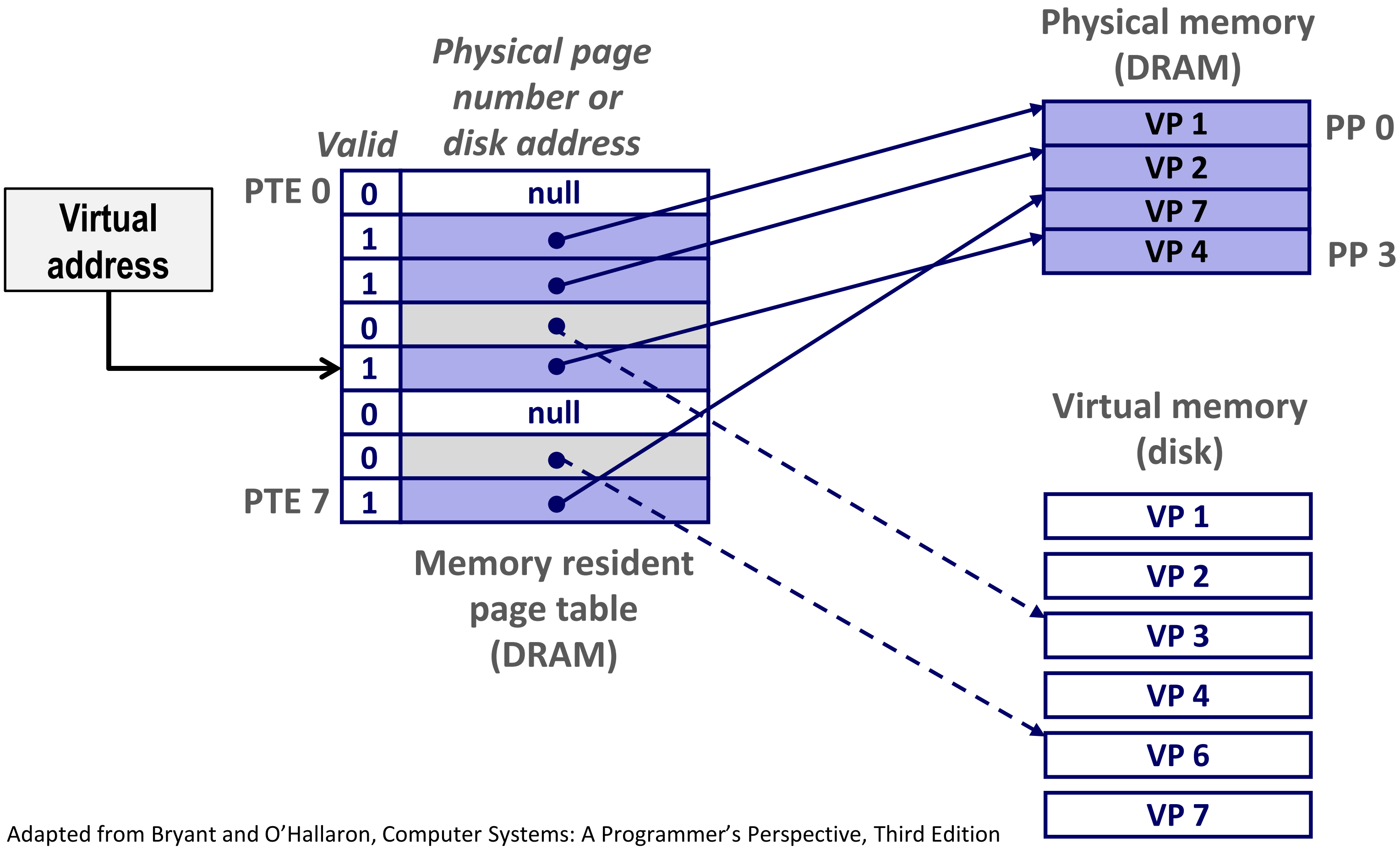
Logically, how does this work?



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Enabling Data Structure: Page Table

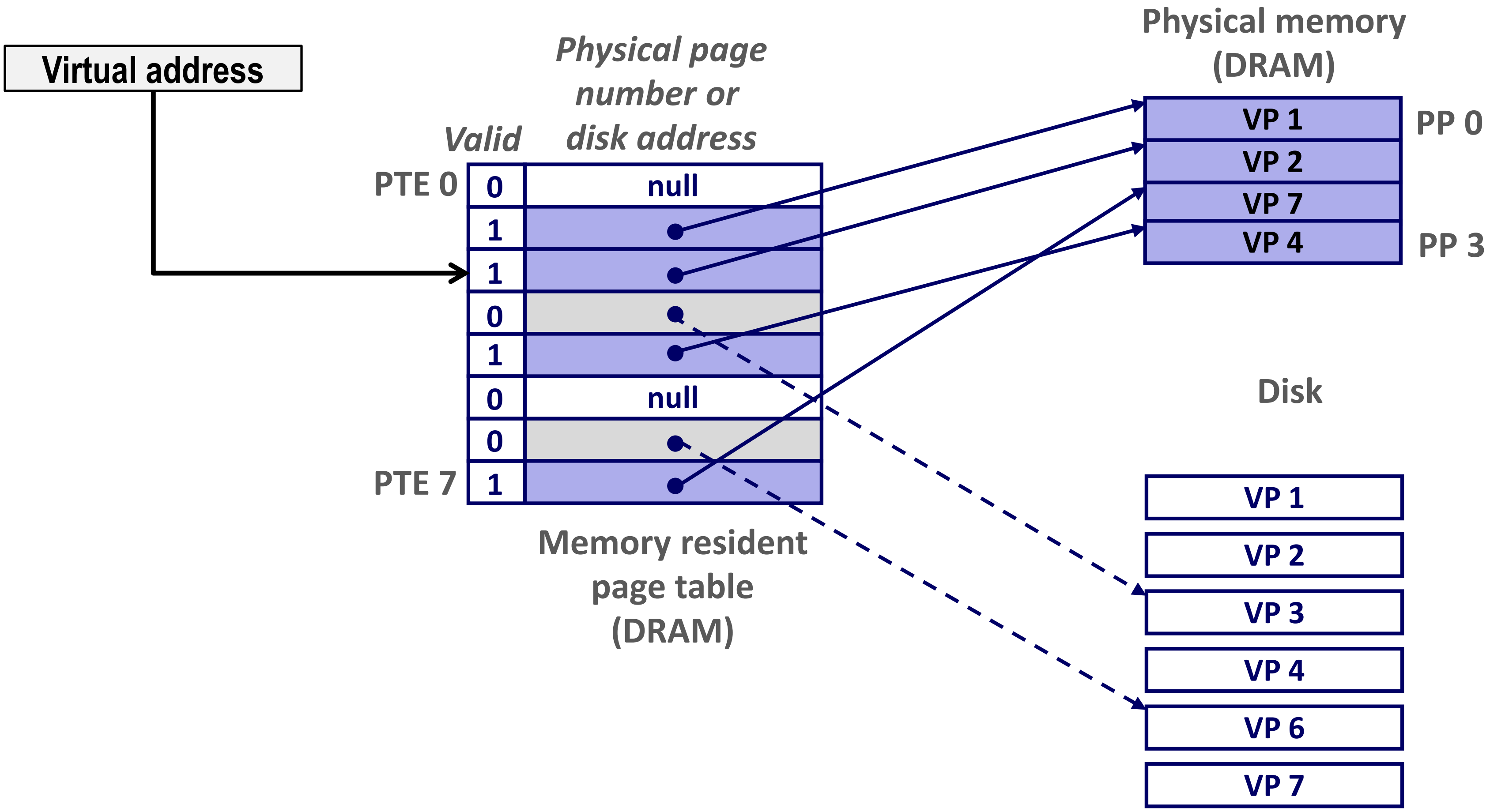
- **A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.**
  - Per-process kernel data structure in DRAM



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Page Hit

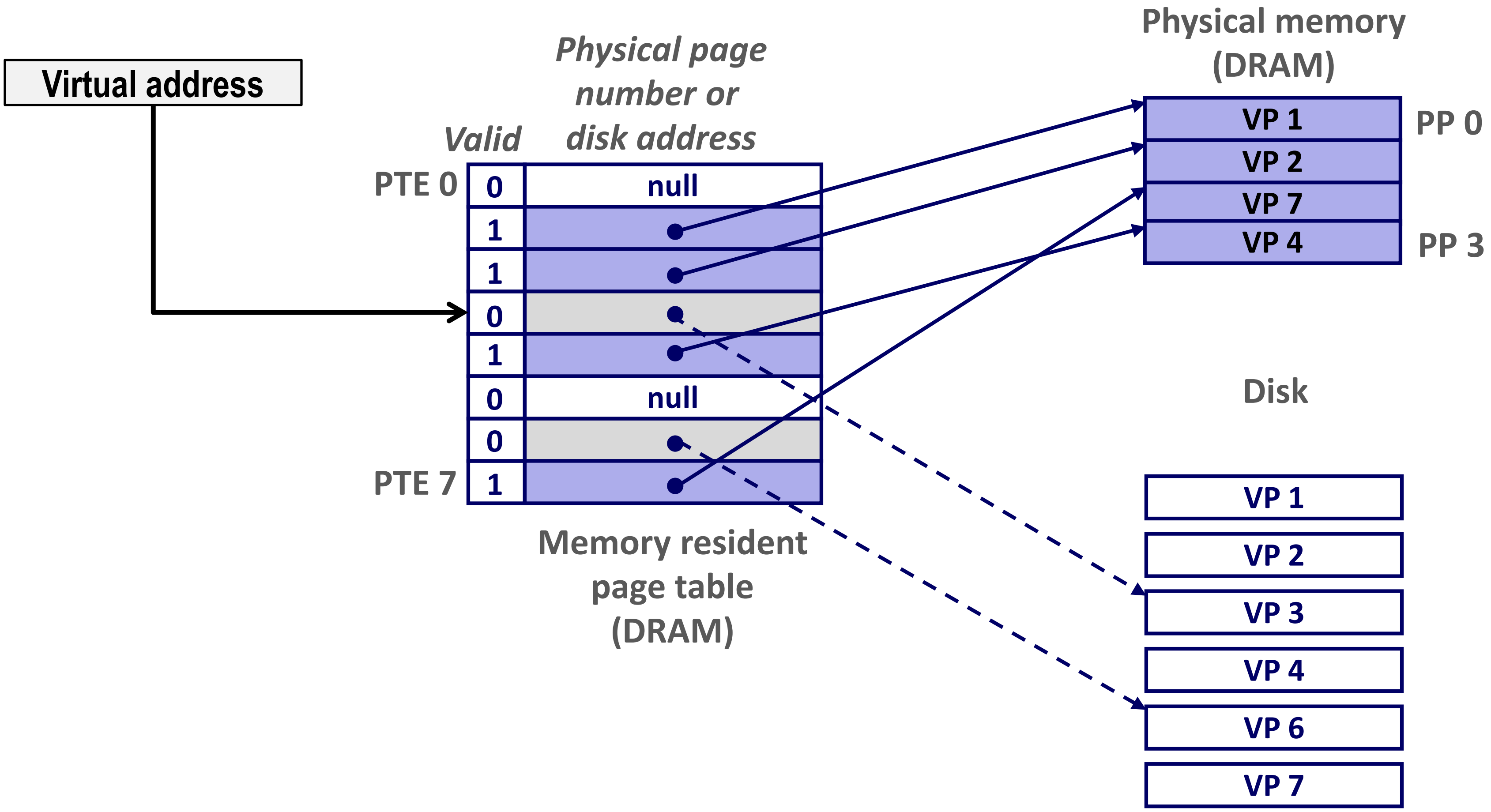
- ***Page hit:*** **reference to VM word that is in physical memory (DRAM cache hit)**



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Page Fault

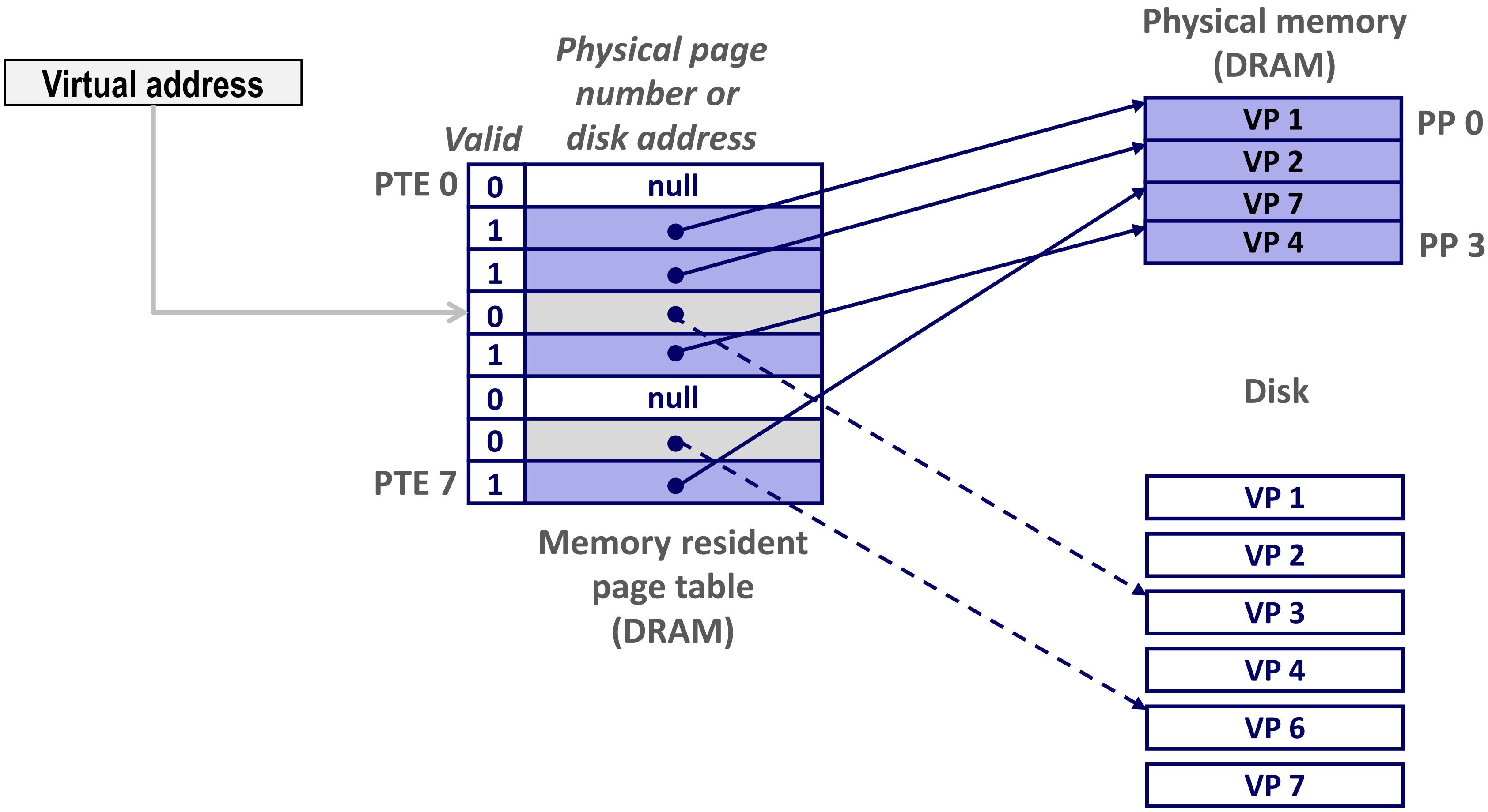
- *Page fault:* reference to VM word that is not in physical memory (DRAM cache miss)



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Handling Page Fault

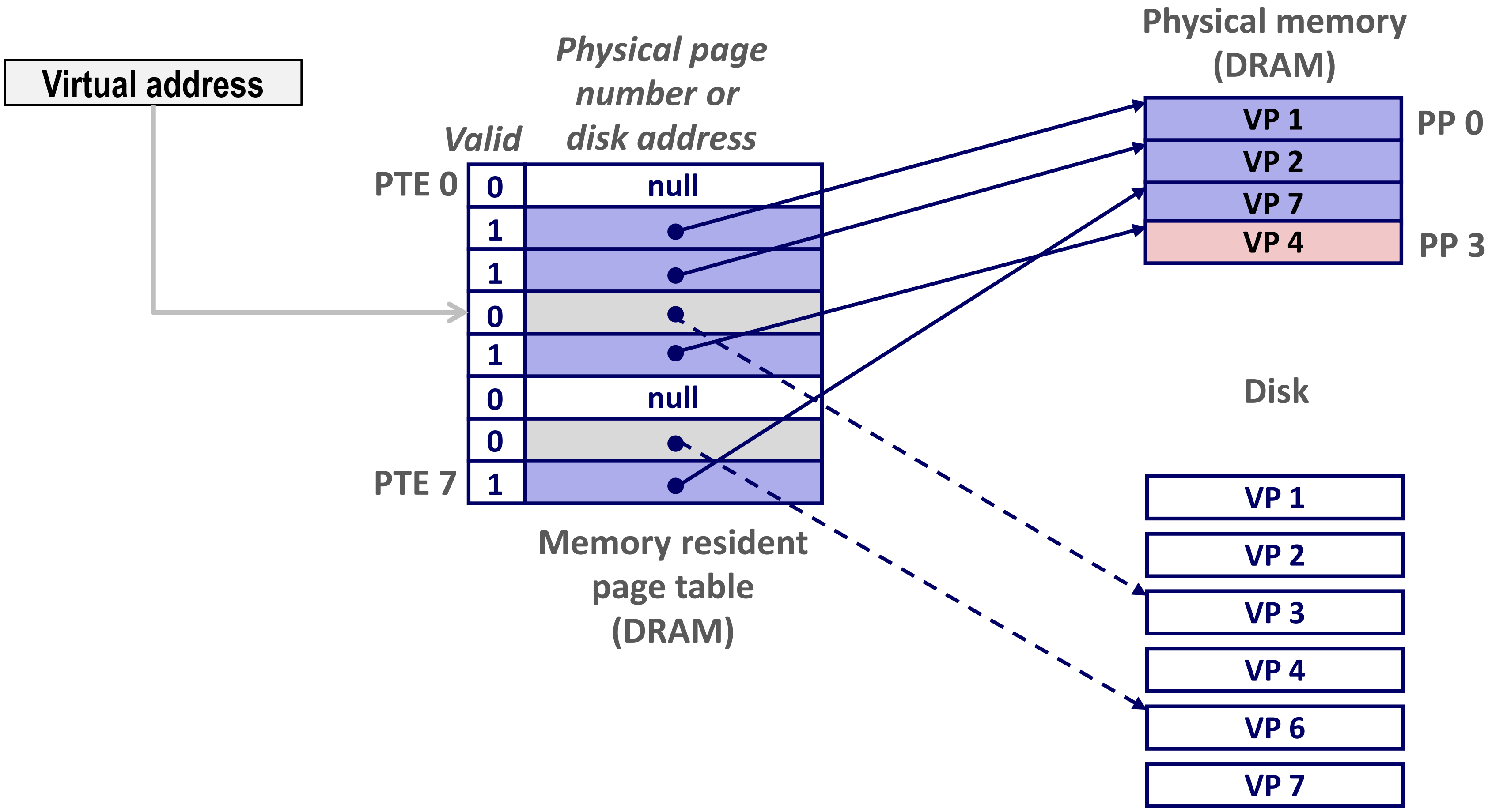
- Page miss causes page fault (an exception)



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Handling Page Fault

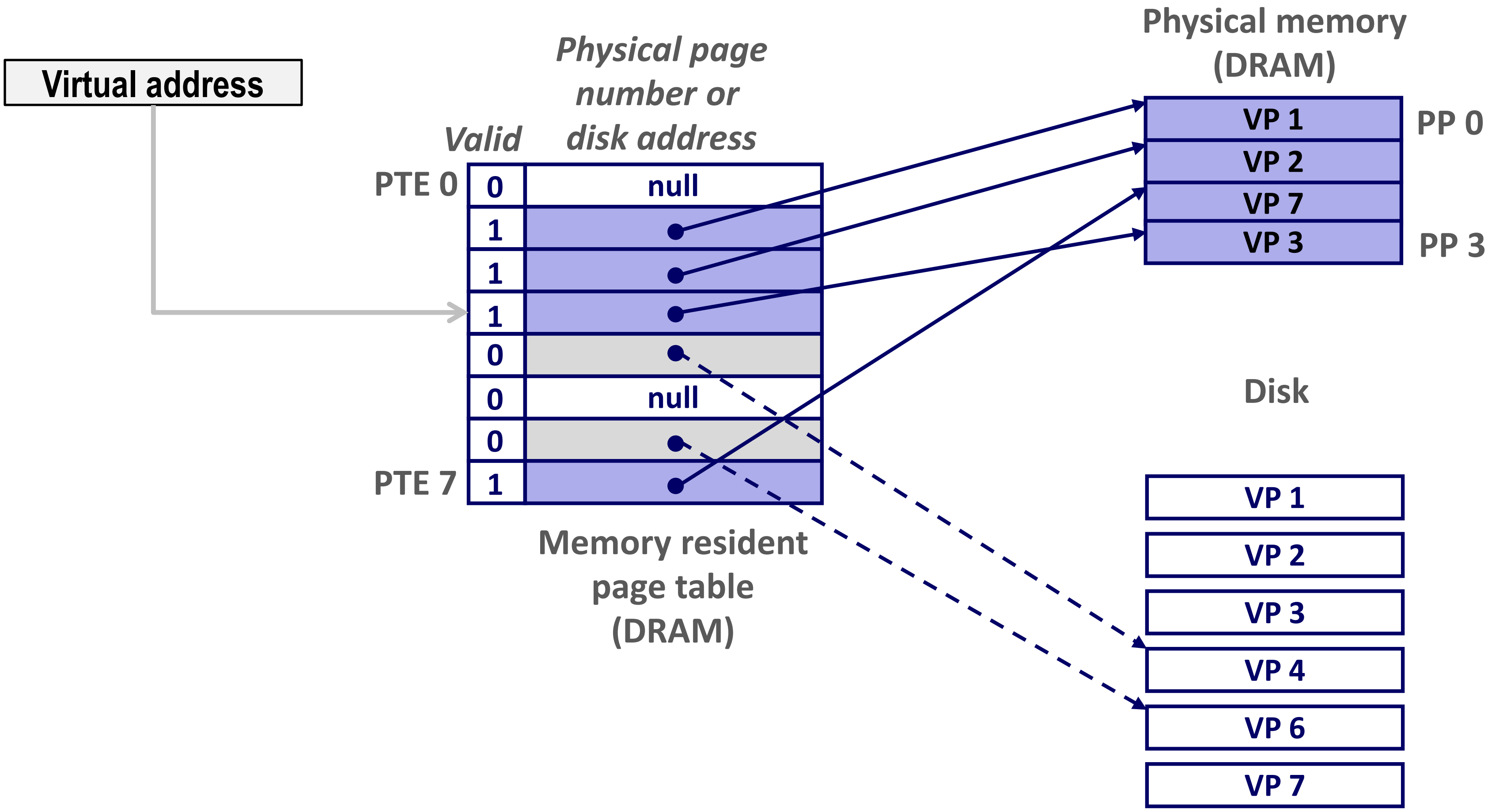
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Handling Page Fault

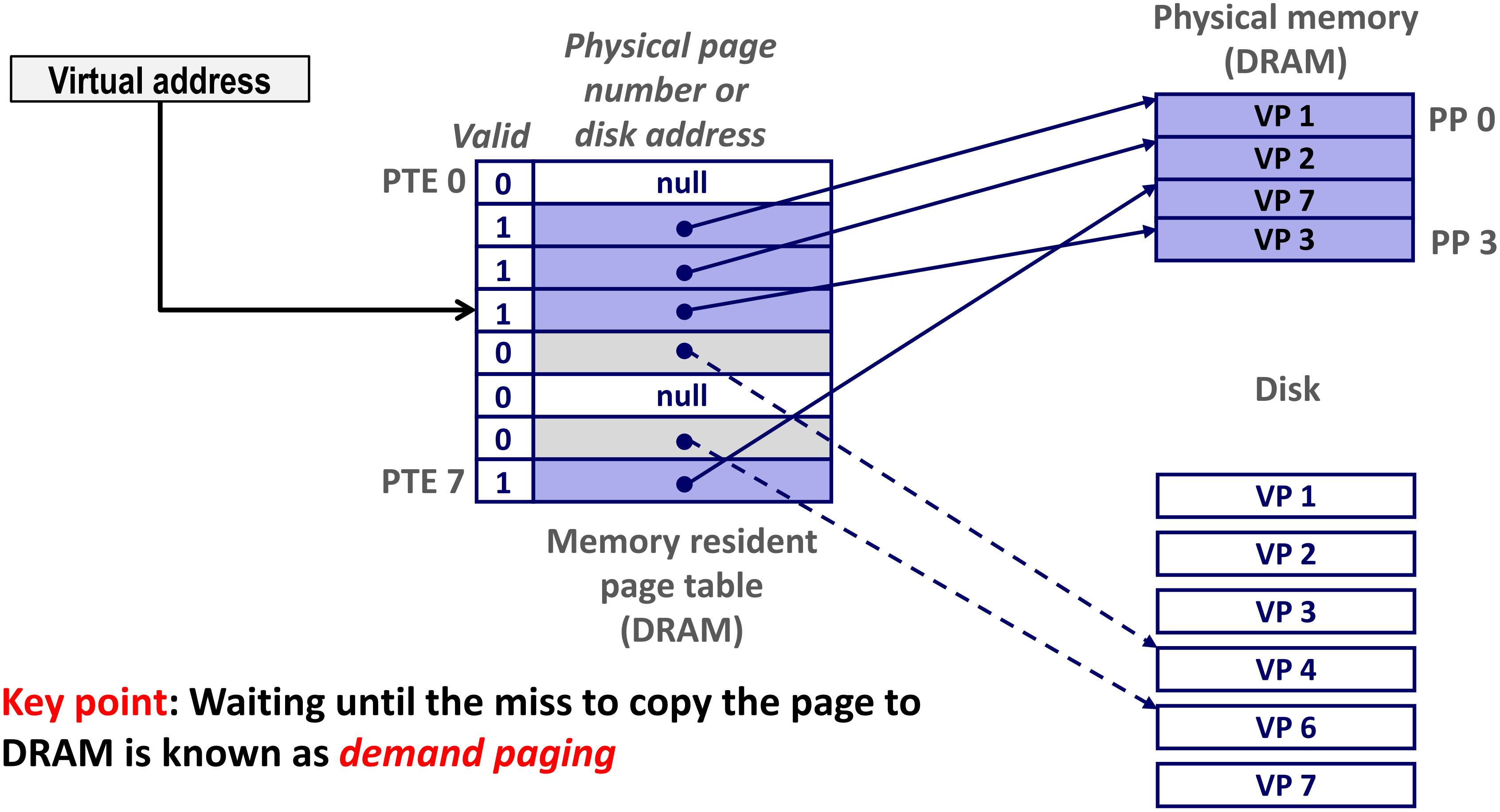
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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!

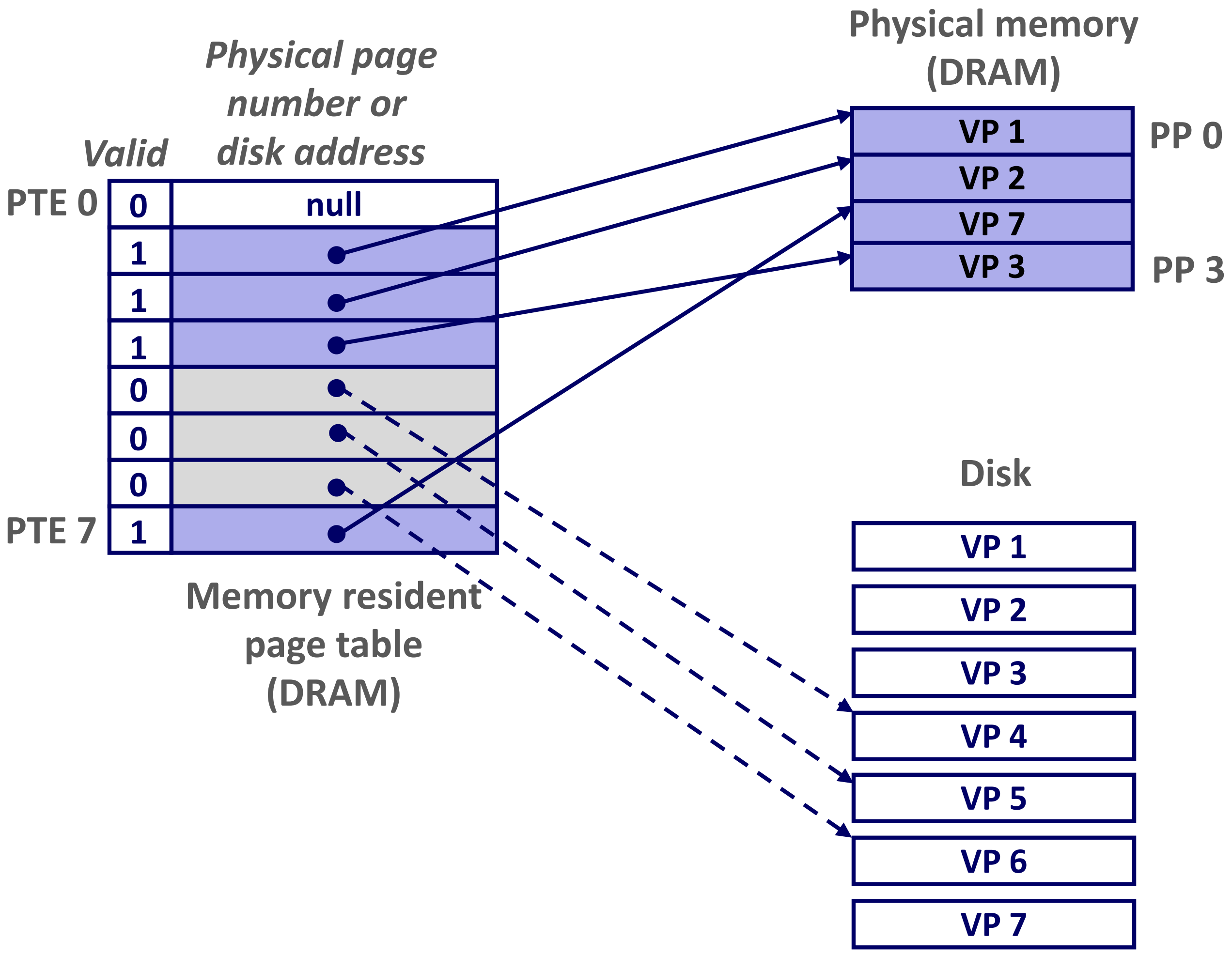


Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

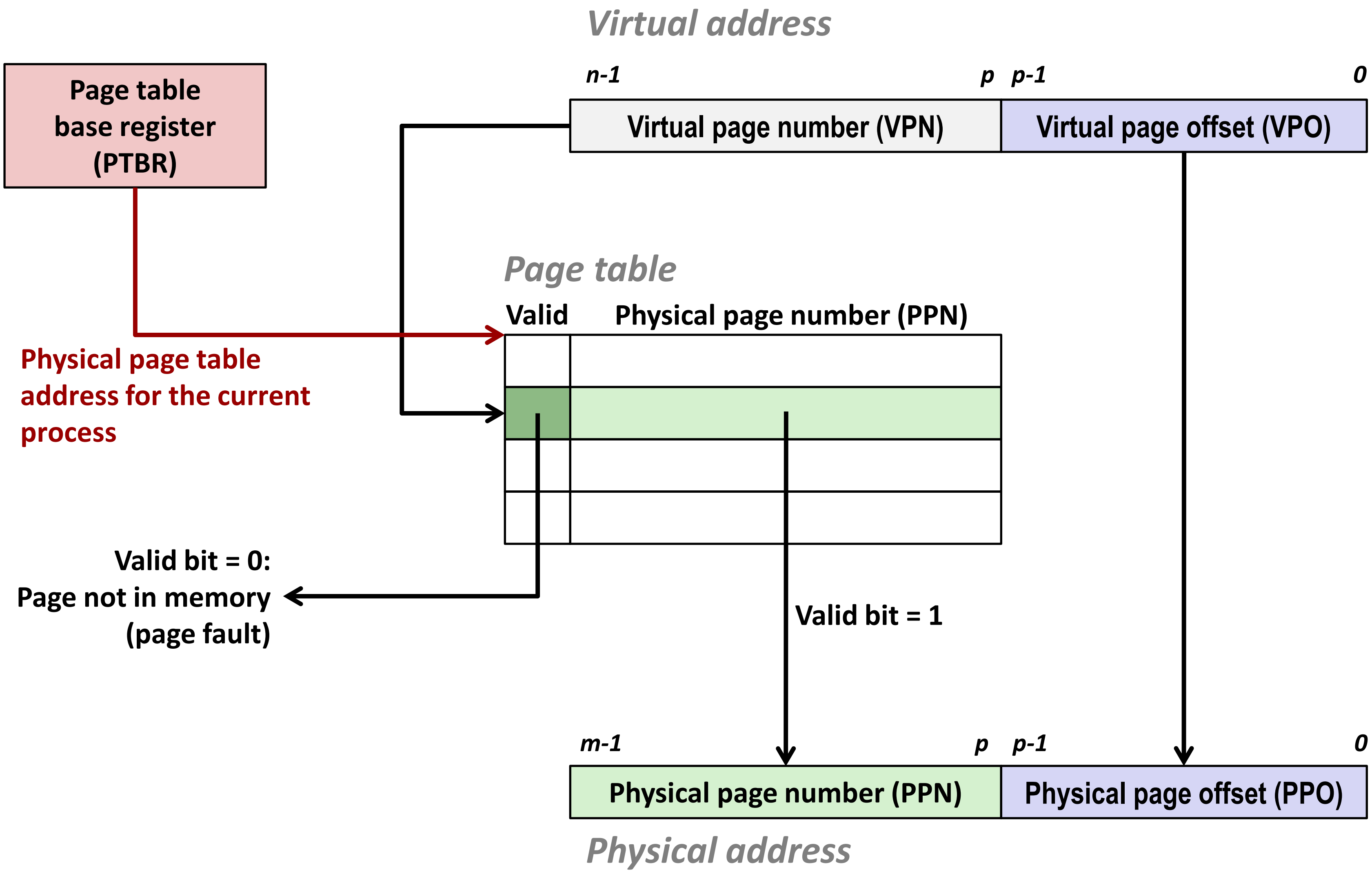
Allocating Pages

- **Allocating a new page (VP 5) of virtual memory.**



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

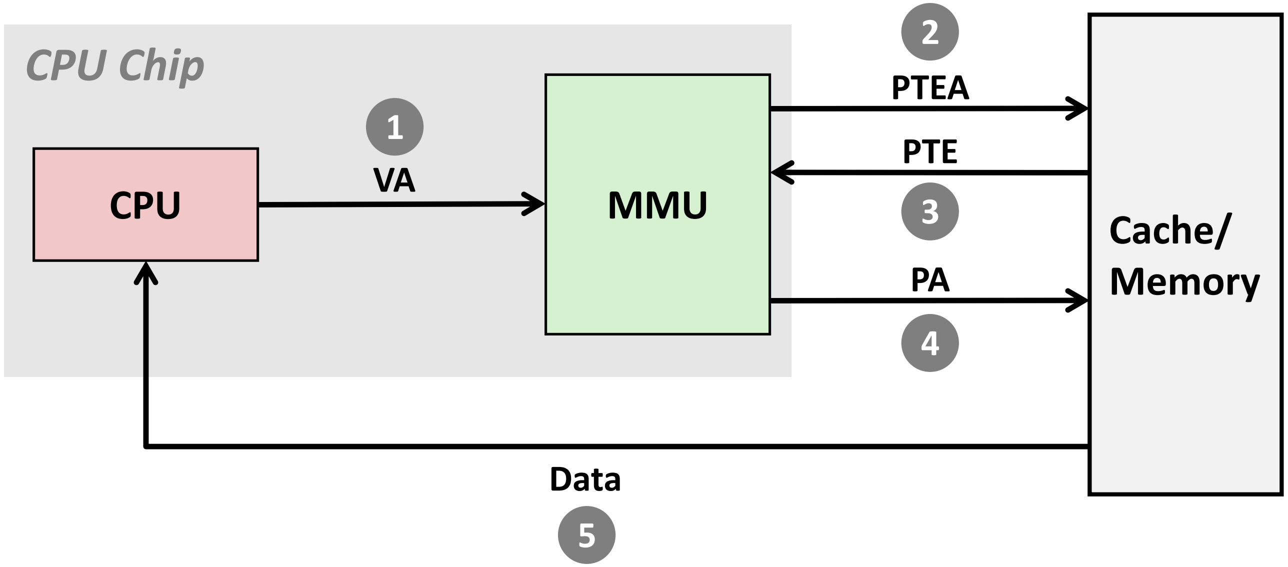
Address Translation With a Page Table



Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR)
Page table
Valid
Physical page number (PPN)
Physical page table address for the current process
Valid bit = 0: Page not in memory (page fault)
Valid bit = 1
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Address Translation: Page Hit



1) Processor sends virtual address to MMU

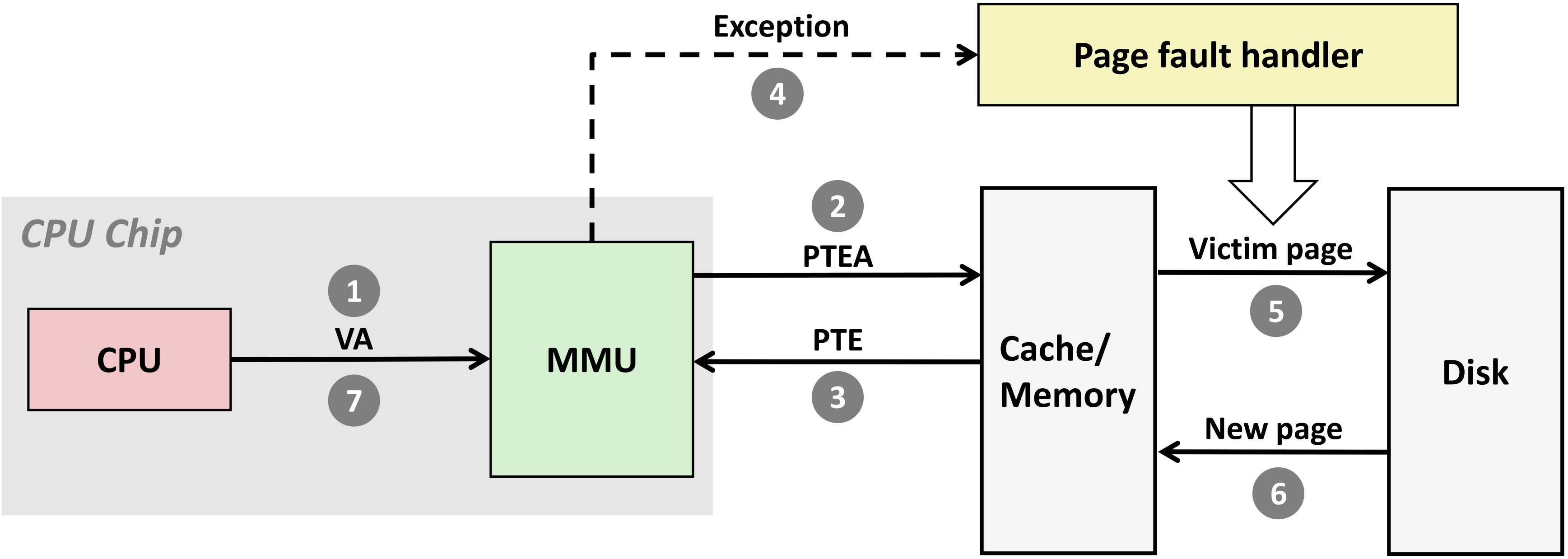
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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)

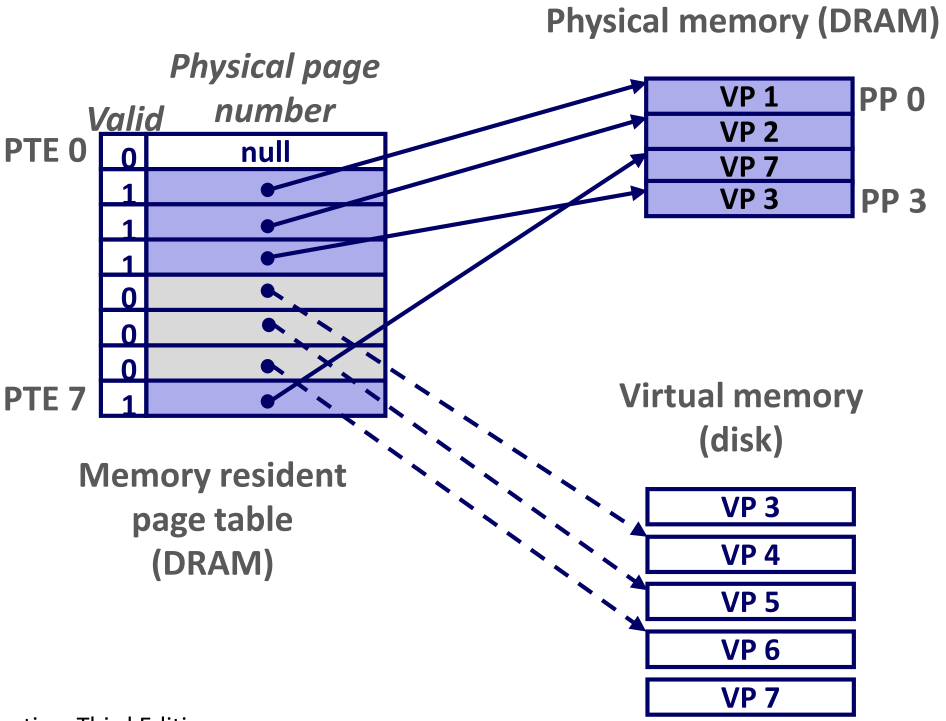
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Simple right? But…

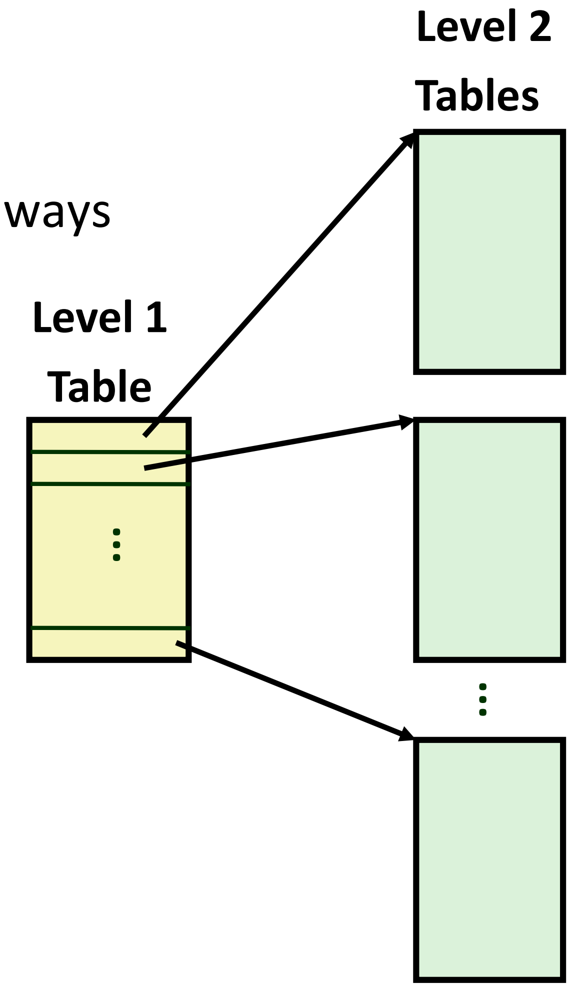
- **Supporting a 48-bit address space is hard…**
  - Each page requires an entry
  - #Entries = Address Space Size / Page Size = 2^48 / 2^12 = 2^36
- **But, the address space is sparse!**
  - No programs *actually* use 2^48 bytes
  - Need a better representation



Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

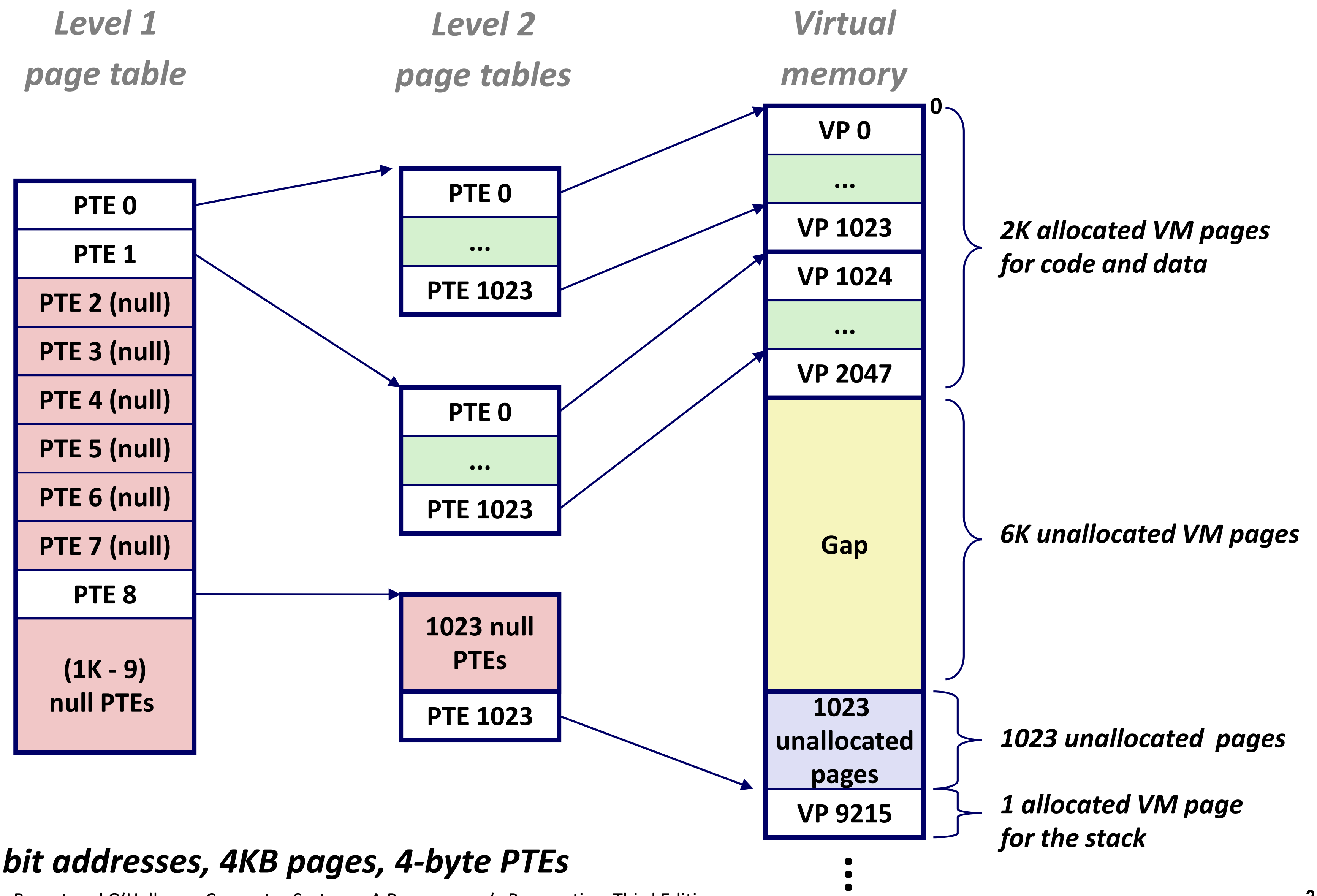
Multi-Level Page Tables

- **Common solution: Multi-level page table**
- **Example: 2-level page table**
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

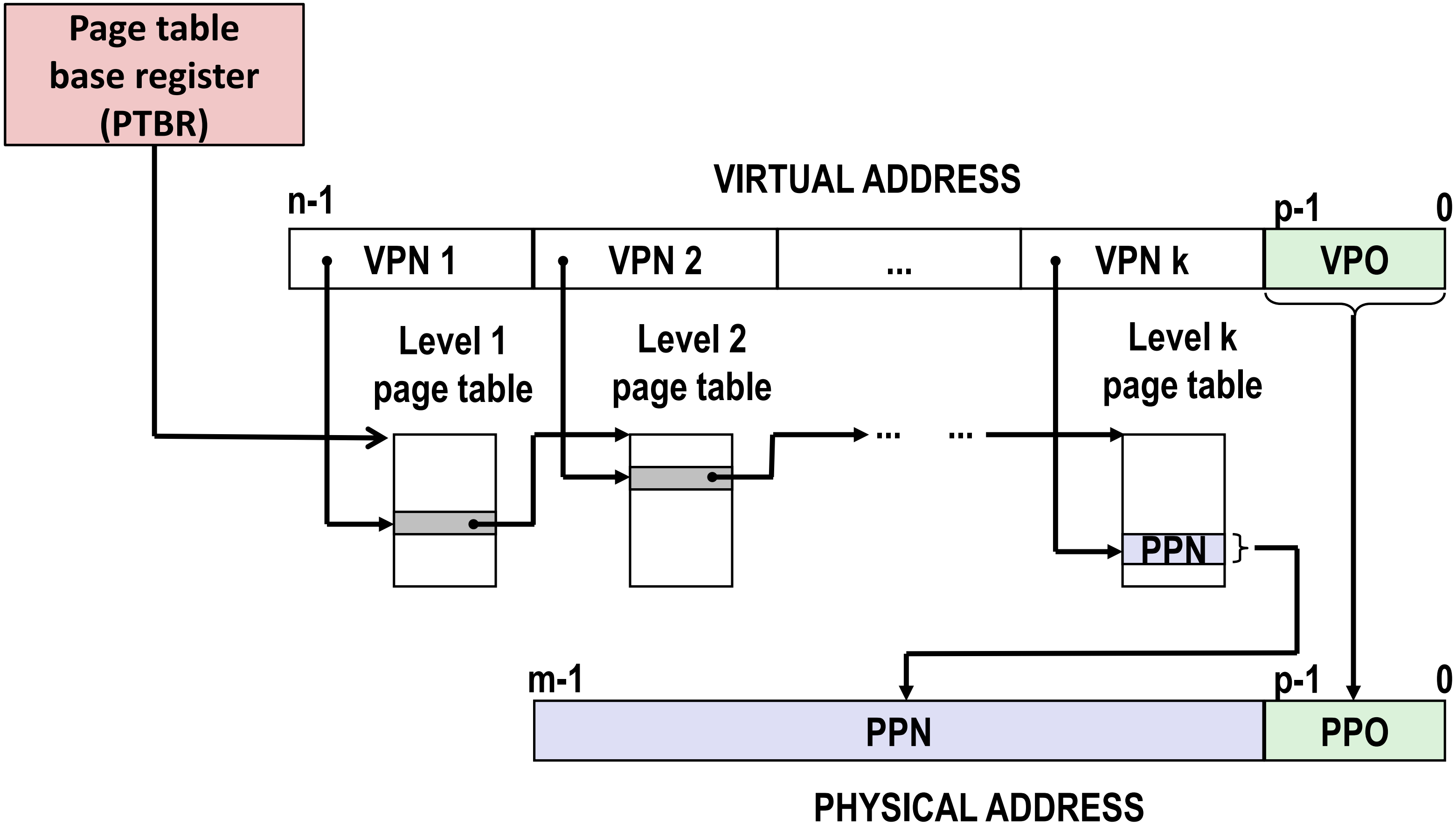
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Translating with a k-level Page Table



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

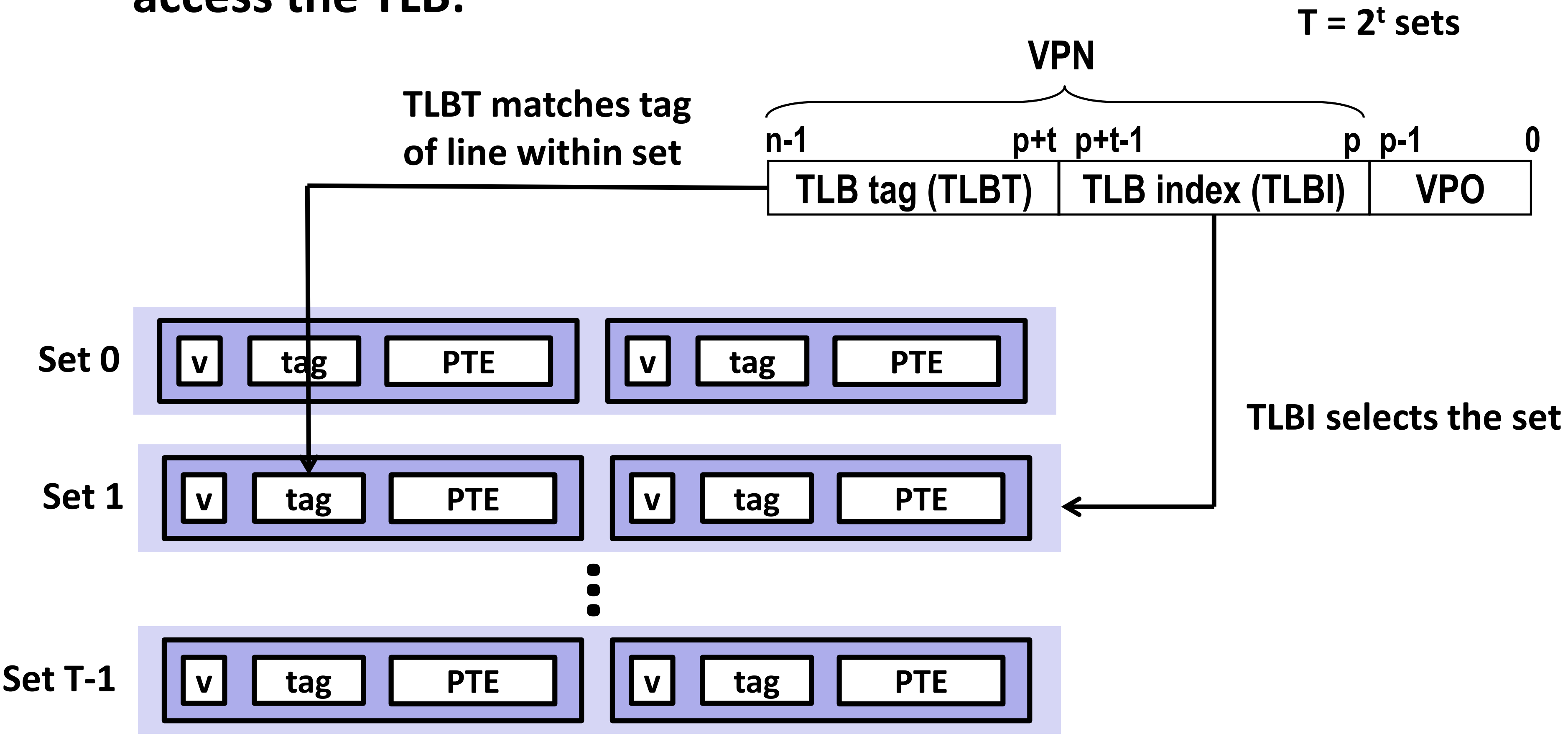
OMG, this is slow!

- **We have to access memory K times for each original memory operation (load/store)!!!!**

- **… We need another Cache!**
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

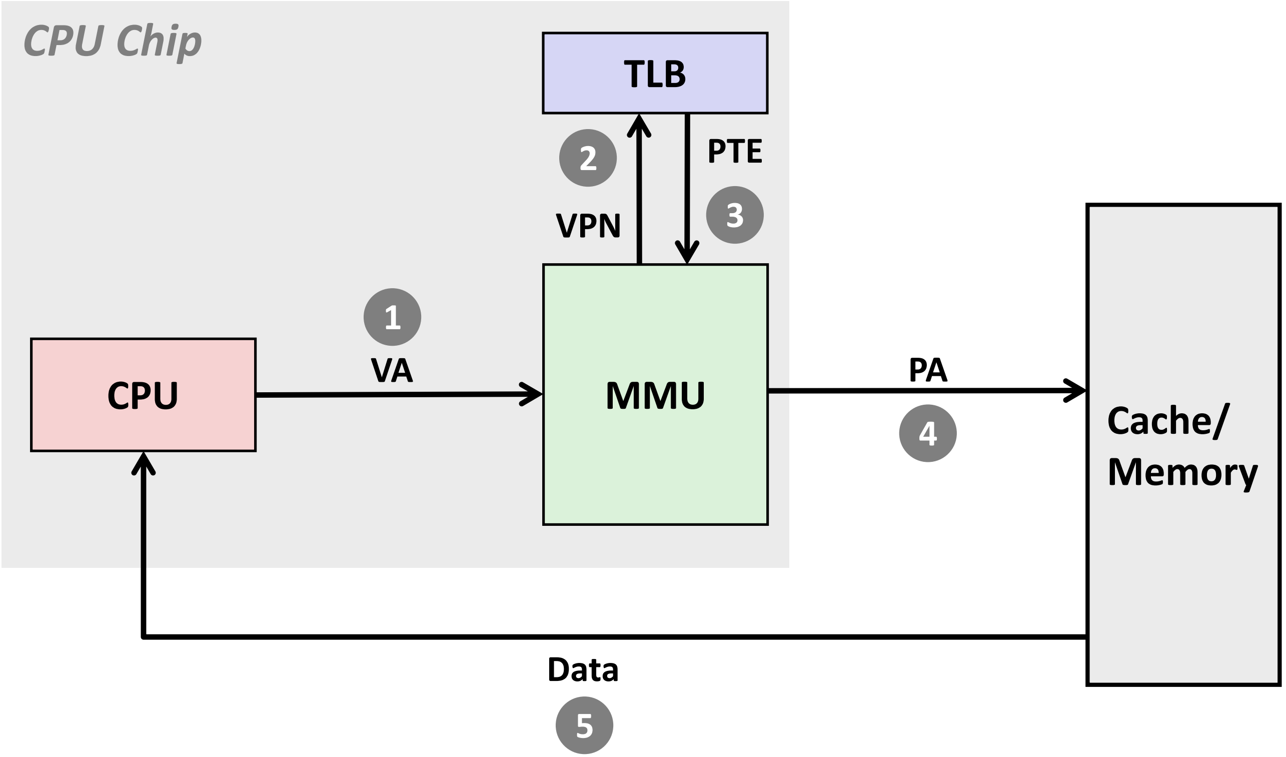
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

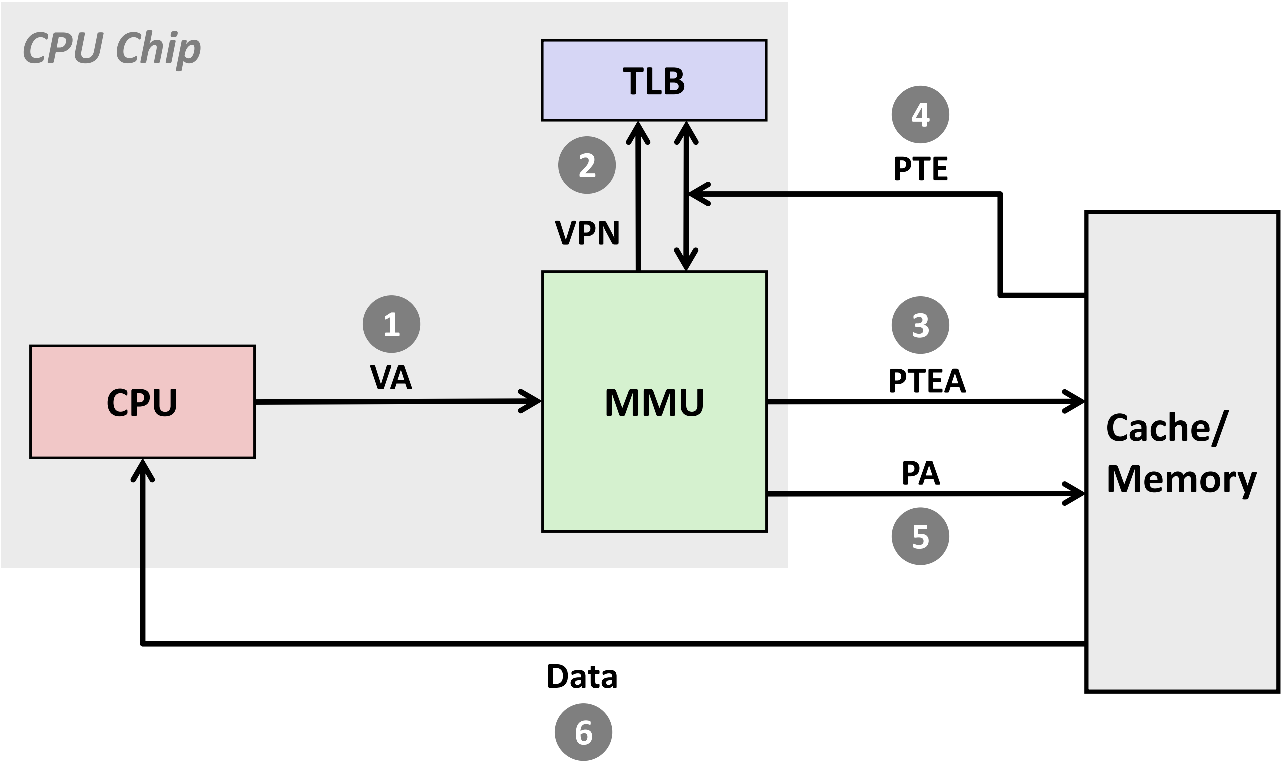
TLB Hit



A TLB hit eliminates a memory access

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

TLB Miss



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

Fortunately, TLB misses are rare. Why?

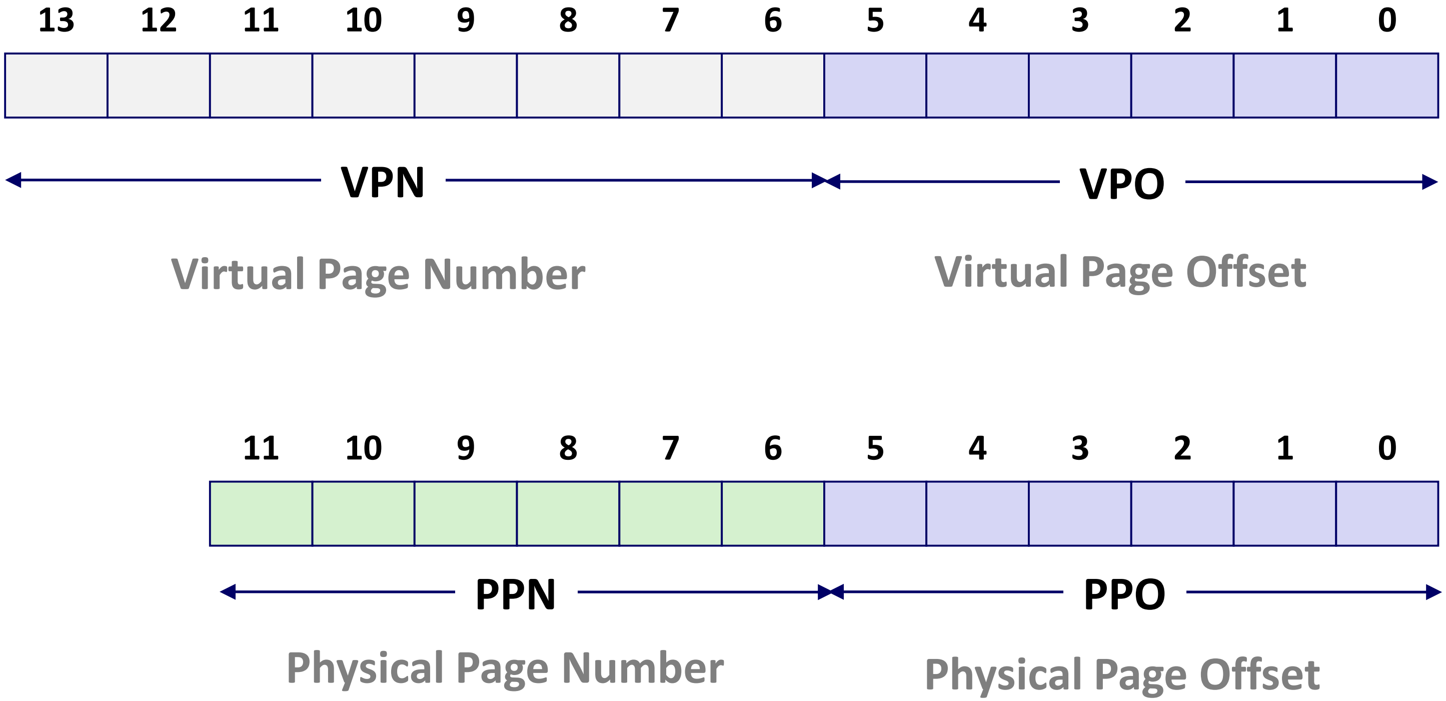
Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Locality to the Rescue Again!

- **Virtual memory seems terribly inefficient, but it 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 compulsory misses
- **If (SUM(working set sizes) > main memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Simple Memory System Example

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



13 12 11 10 9 8 7 6 5 4 3 2 1 0

VPN

VPO

Virtual Page Number

Virtual Page Offset

11 10 9 8 7 6 5 4 3 2 1 0

PPN

PPO

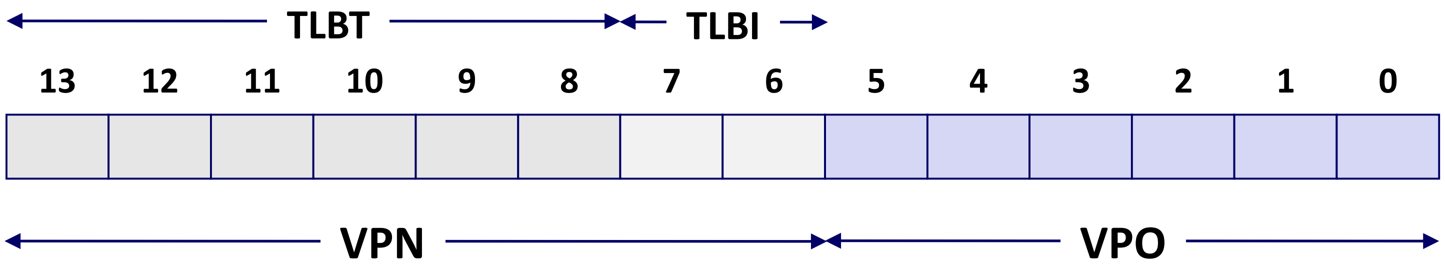
Physical Page Number

Physical Page Offset

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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

Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

2. Simple Memory System Page Table

Only show first 16 entries (out of 256)

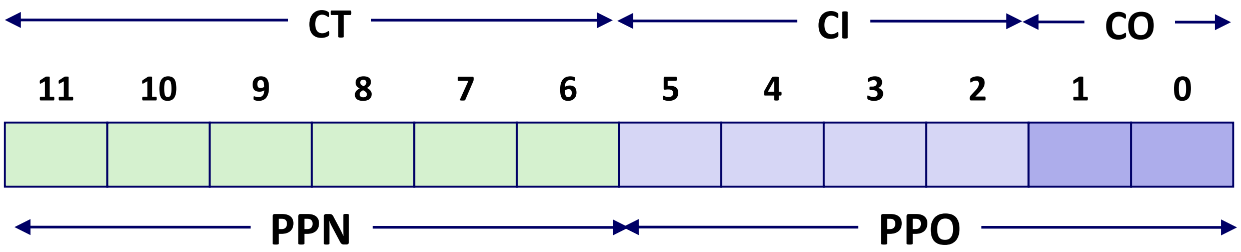
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

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

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

3. Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped



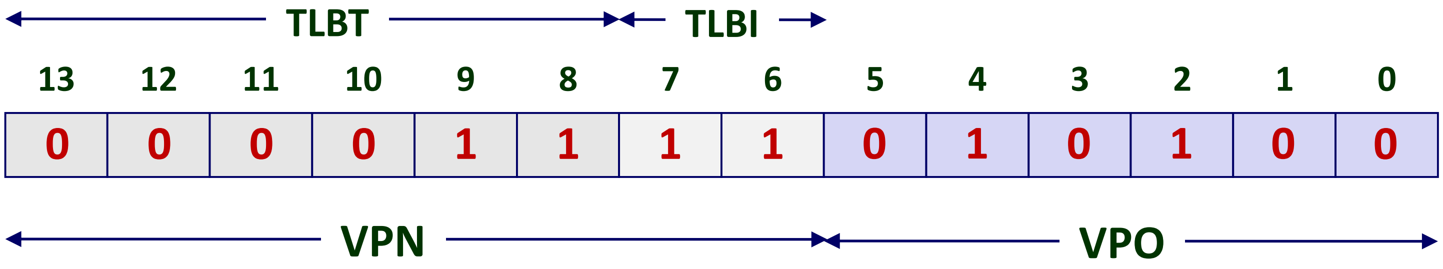
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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	–	–	–	–

Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

Address Translation Example #1

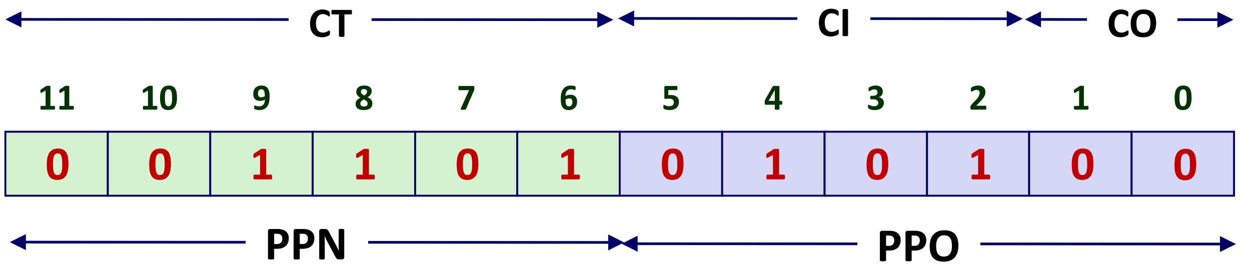
Virtual Address: 0x03D4



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

Physical Address



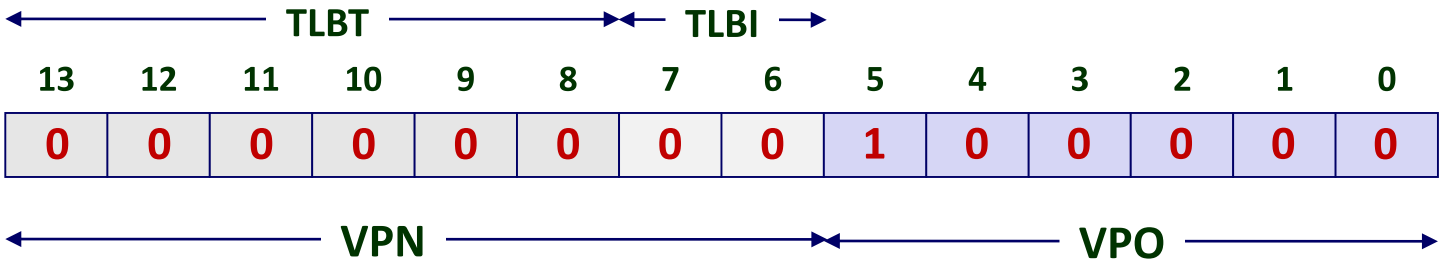
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

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

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Address Translation Example #2

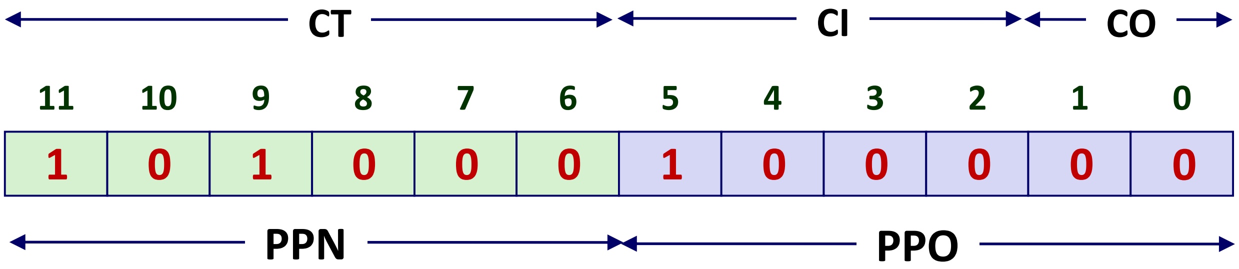
Virtual Address: 0x0020



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

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

Physical Address



11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Why Virtual Memory (VM)?

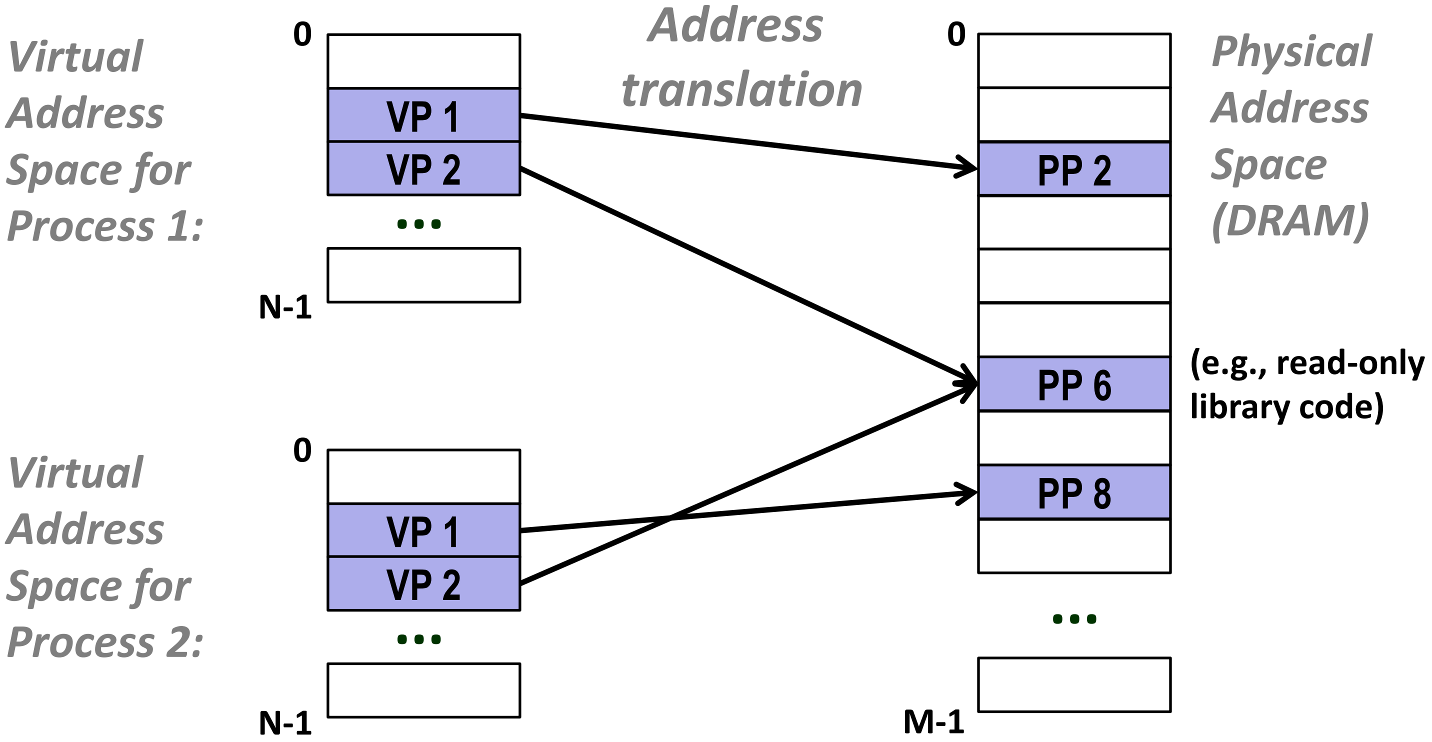
- **Uses main memory efficiently**
  - Use DRAM as a cache for parts of a virtual address space

- **Simplifies memory management**
  - Each process gets the same uniform linear address space

- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code

VM as a Tool for Memory Management

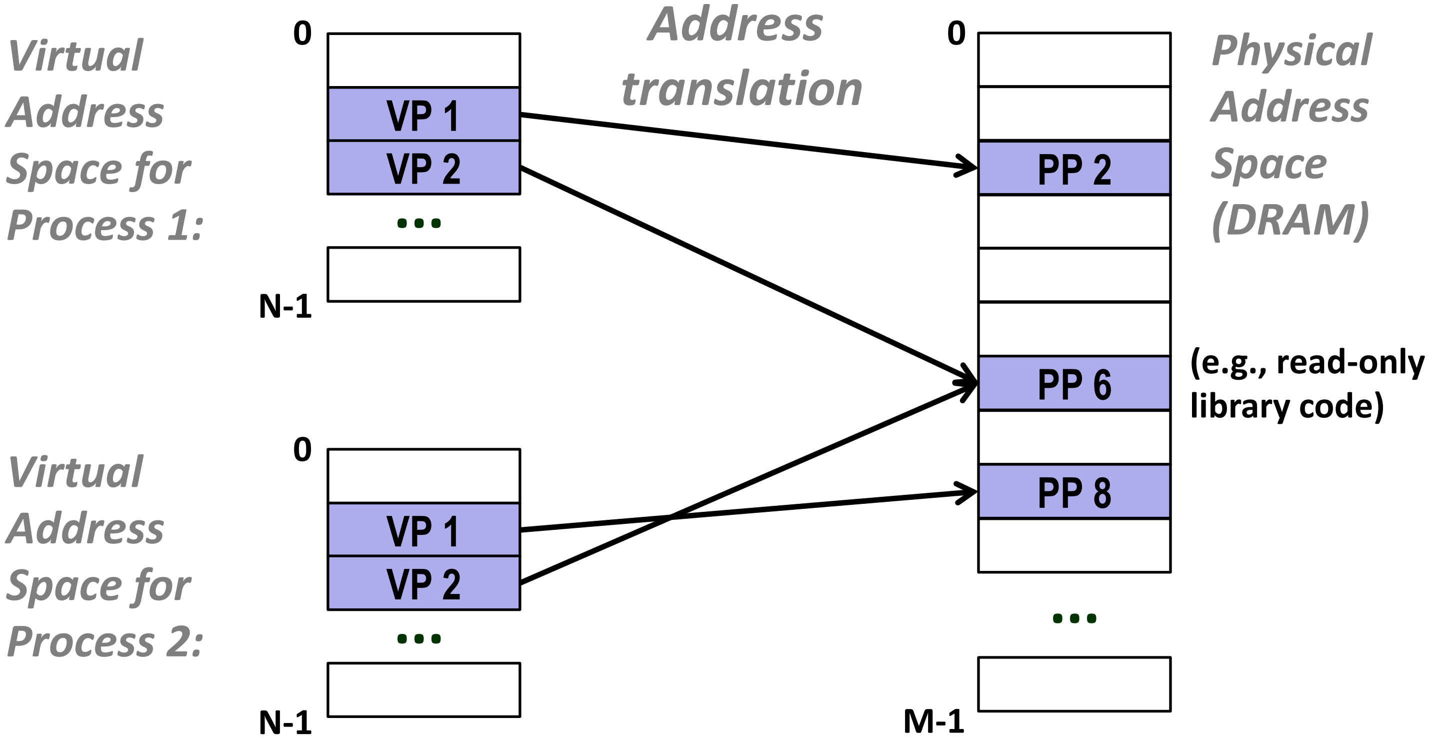
- **Key idea: each process has its own virtual address space**
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well-chosen mappings can improve locality


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

VM as a Tool for Memory Management

- **Simplifying 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)

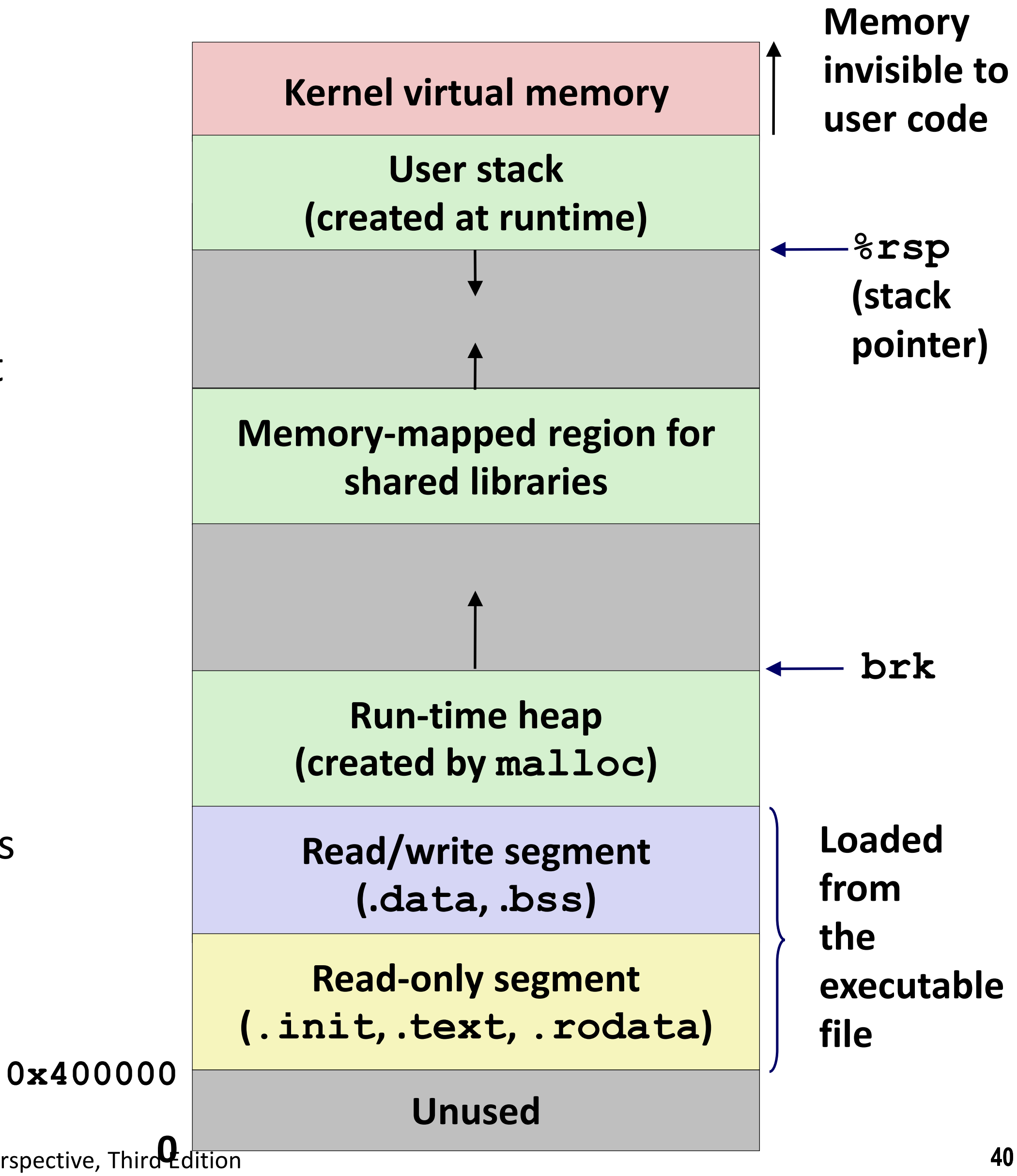

Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.

- **Loading**
  - **`execve`** allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - The **`.text`** and **`.data`** sections are copied, page by page, on demand by the virtual memory system

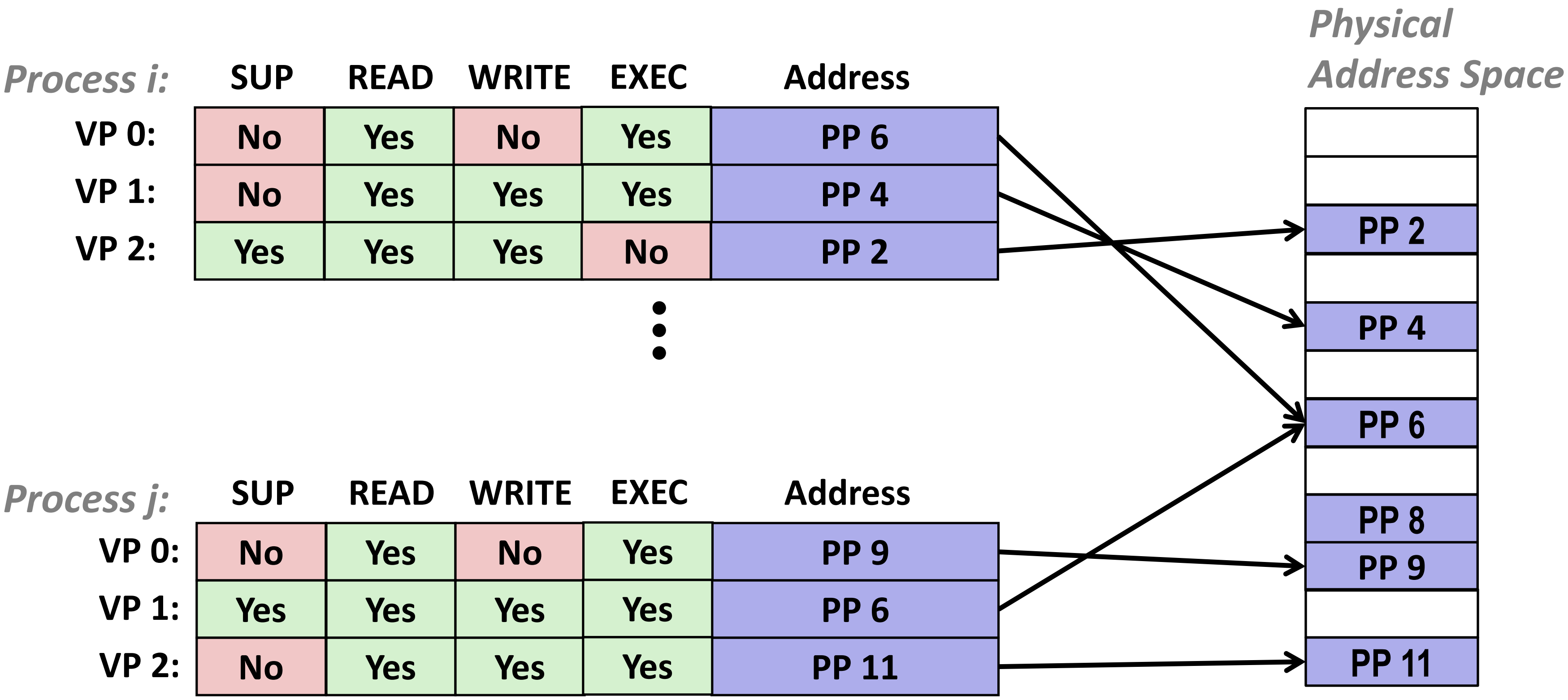


Memory invisible to user code
Kernel virtual memory
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
Unused
0

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access



Process i:

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

•
•
•

Process j:

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

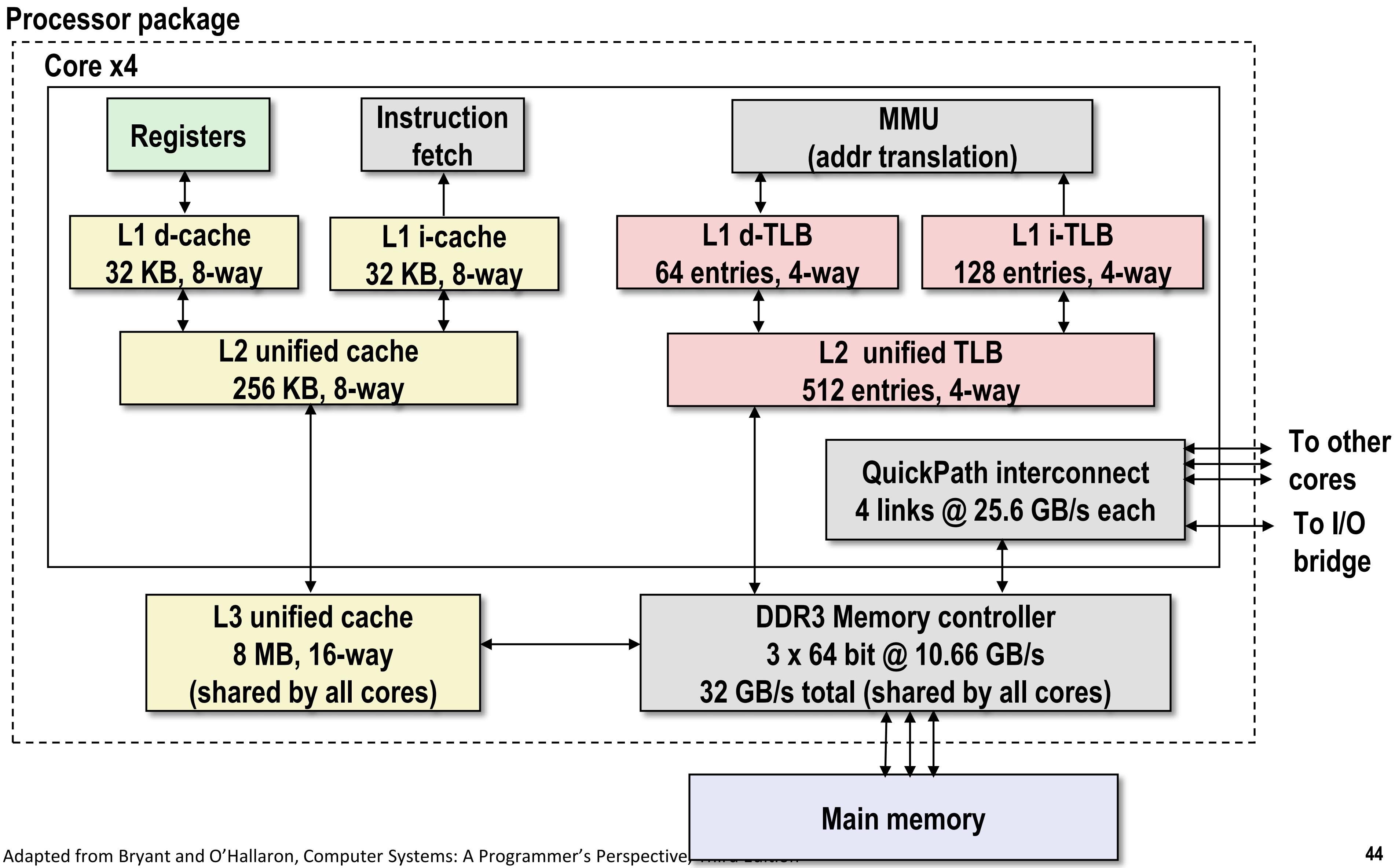
Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

Summary

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

Intel Core i7 Memory System

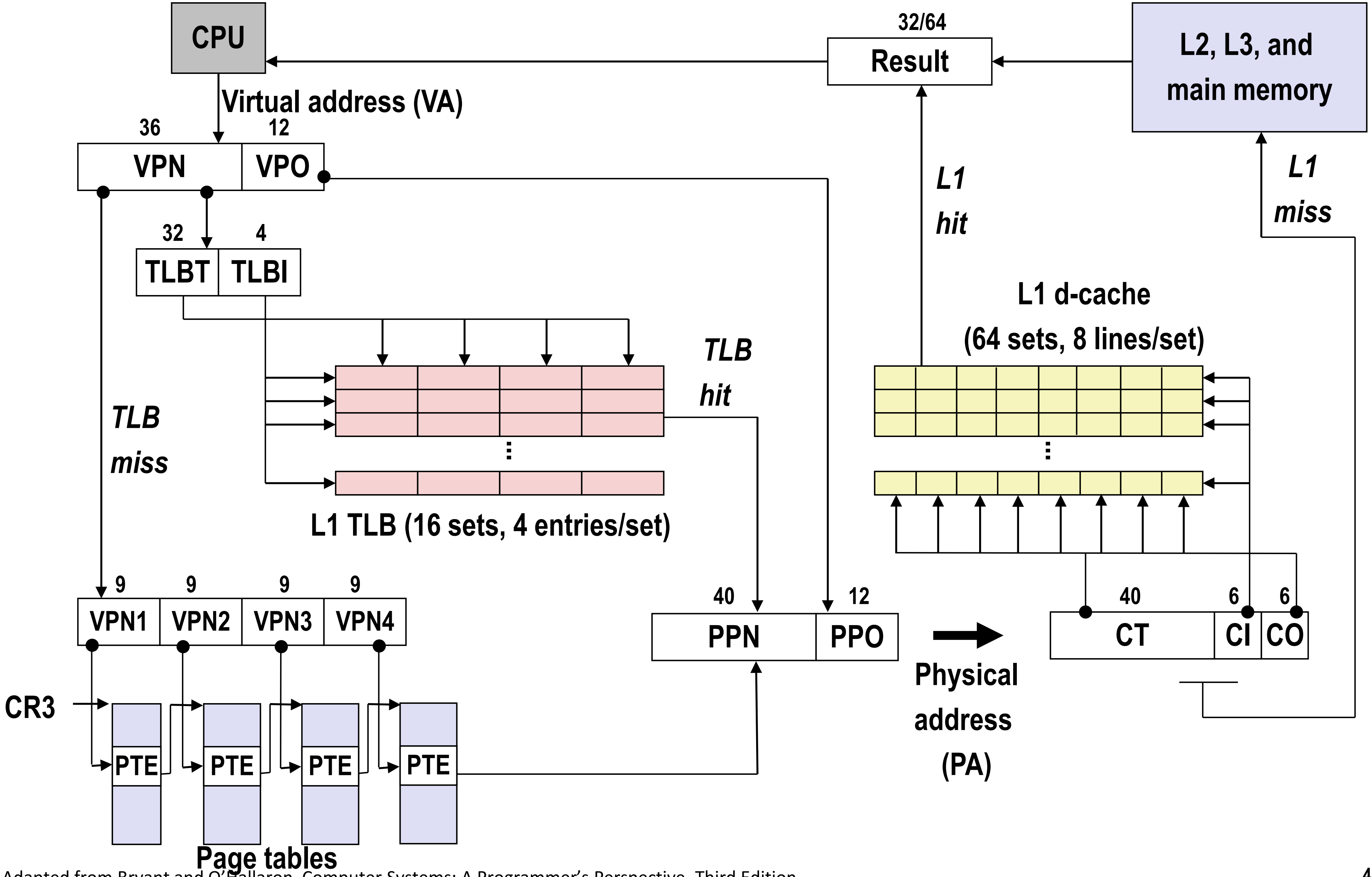

Processor package
Core x4
Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L1 i-cache 32 KB, 8-way
L1 d-TLB 64 entries, 4-way
L1 i-TLB 128 entries, 4-way
L2 unified cache 256 KB, 8-way
L2 unified TLB 512 entries, 4-way
QuickPath interconnect 4 links @ 25.6 GB/s each
To other cores
To I/O bridge
L3 unified cache 8 MB, 16-way (shared by all cores)
DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s 32 GB/s total (shared by all cores)
Main memory

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Review of Symbols

- **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (bytes)
- **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

End-to-end Core i7 Address Translation



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Core i7 Level 1-3 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Core i7 Level 4 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1

Available for OS (page location on disk)	P=0

Each entry references a 4K child page. Significant fields:

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

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

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

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

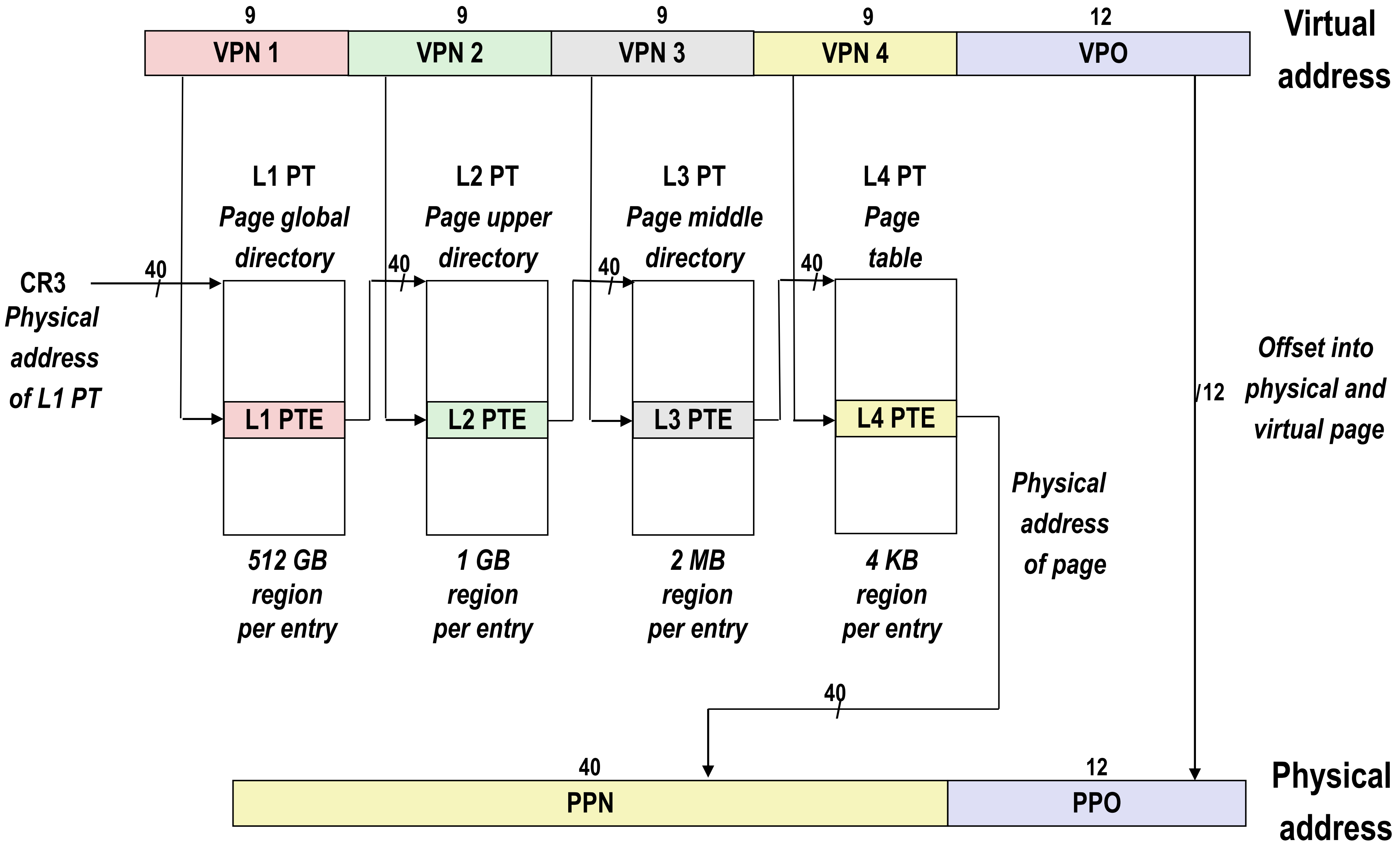
D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

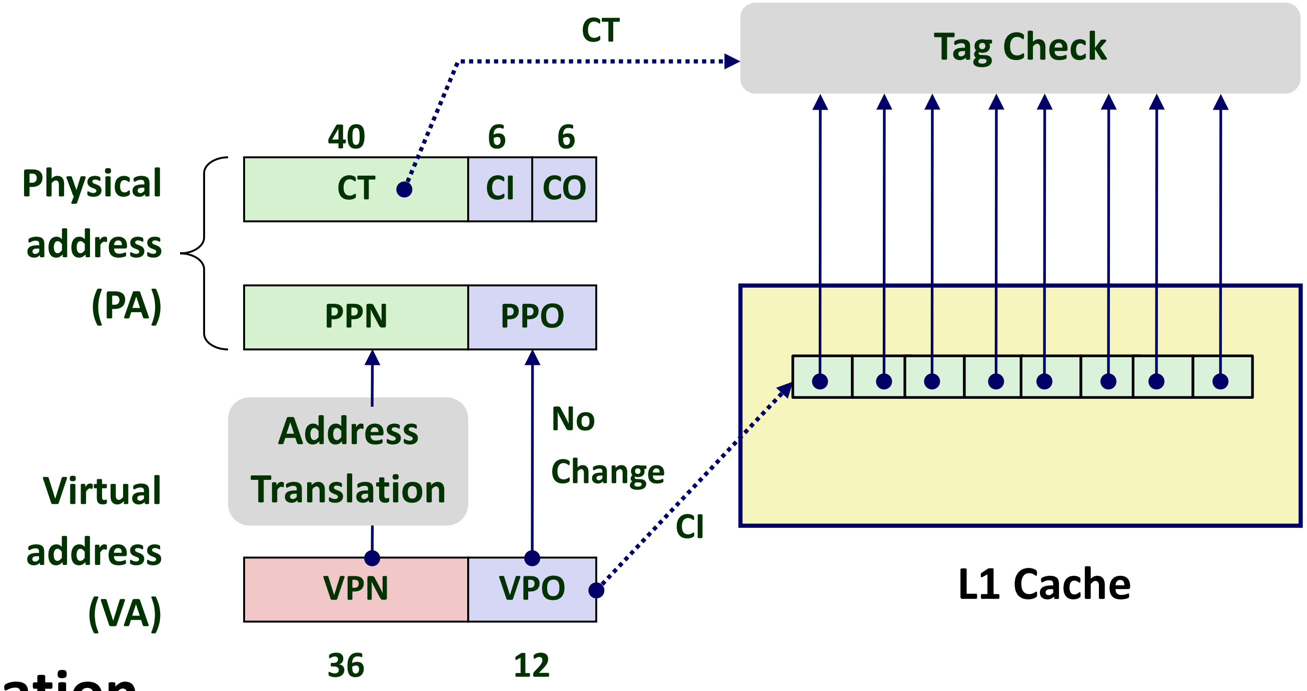
Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Core i7 Page Table Translation



Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

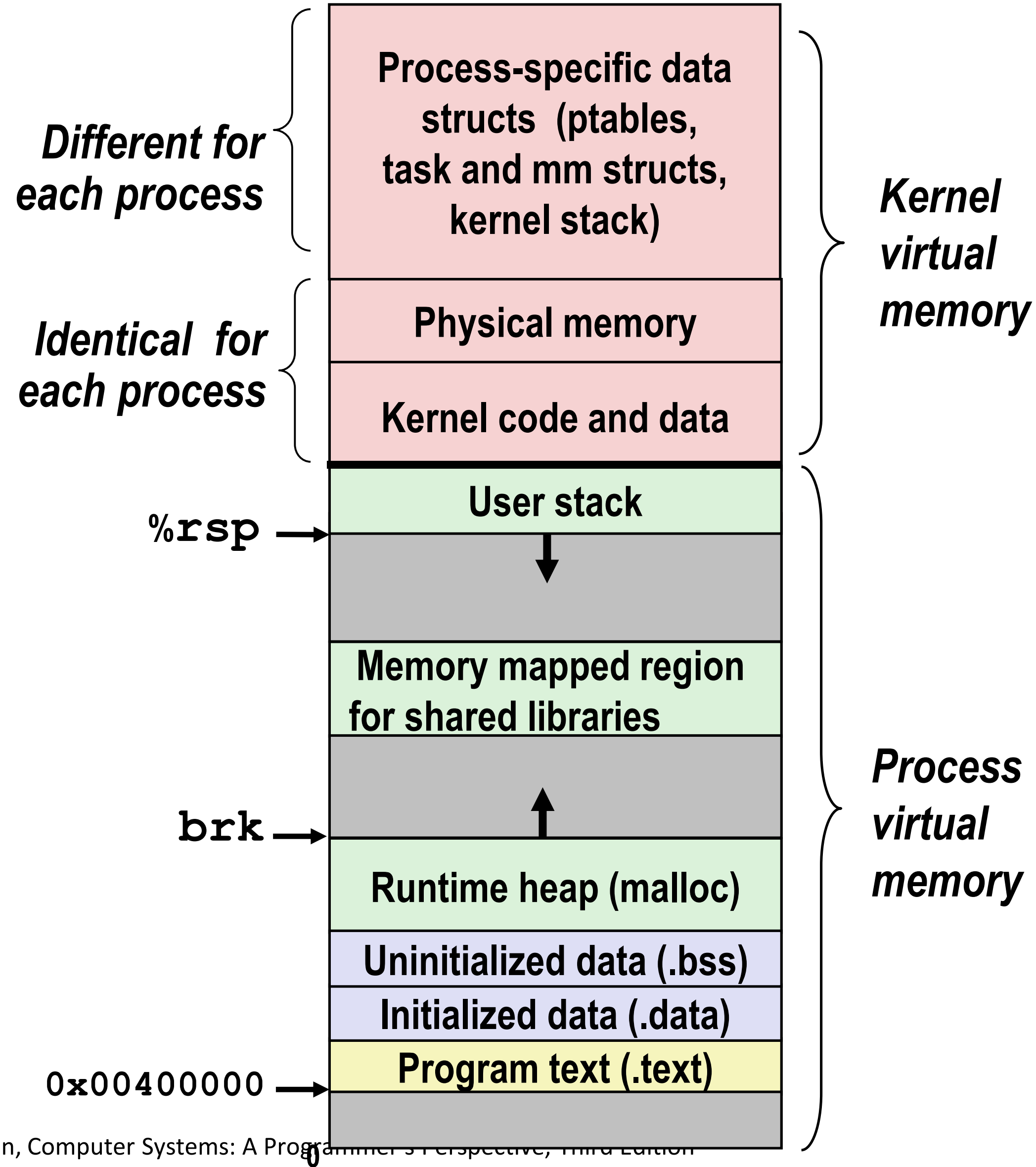
Cute Trick for Speeding Up L1 Access



- **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

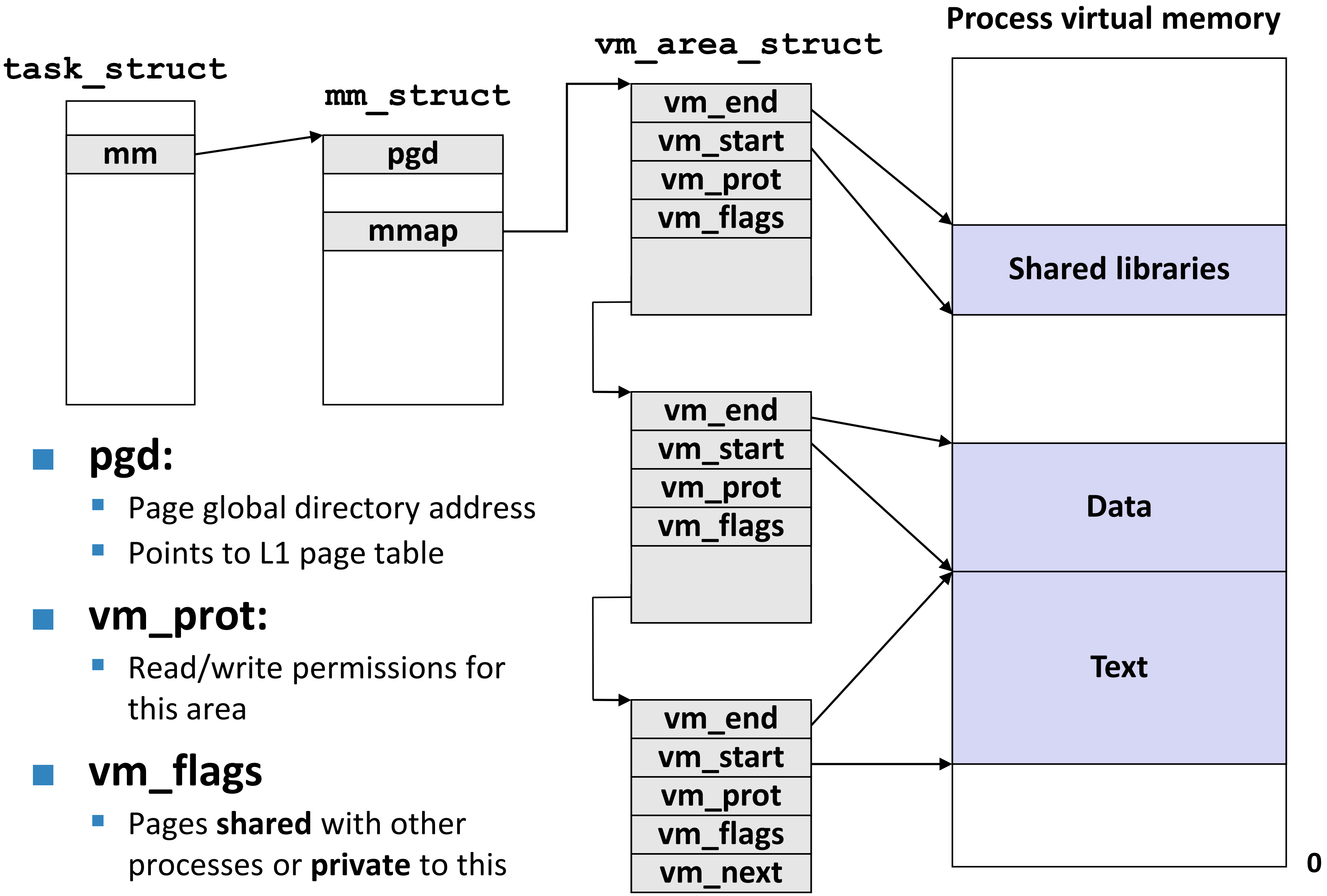
Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Virtual Address Space of a Linux Process


Different for each process
Identical for each process
Process-specific data structs (ptables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%rsp
Memory mapped region for shared libraries
brk
Runtime heap (malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x00400000
0
Process virtual memory

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

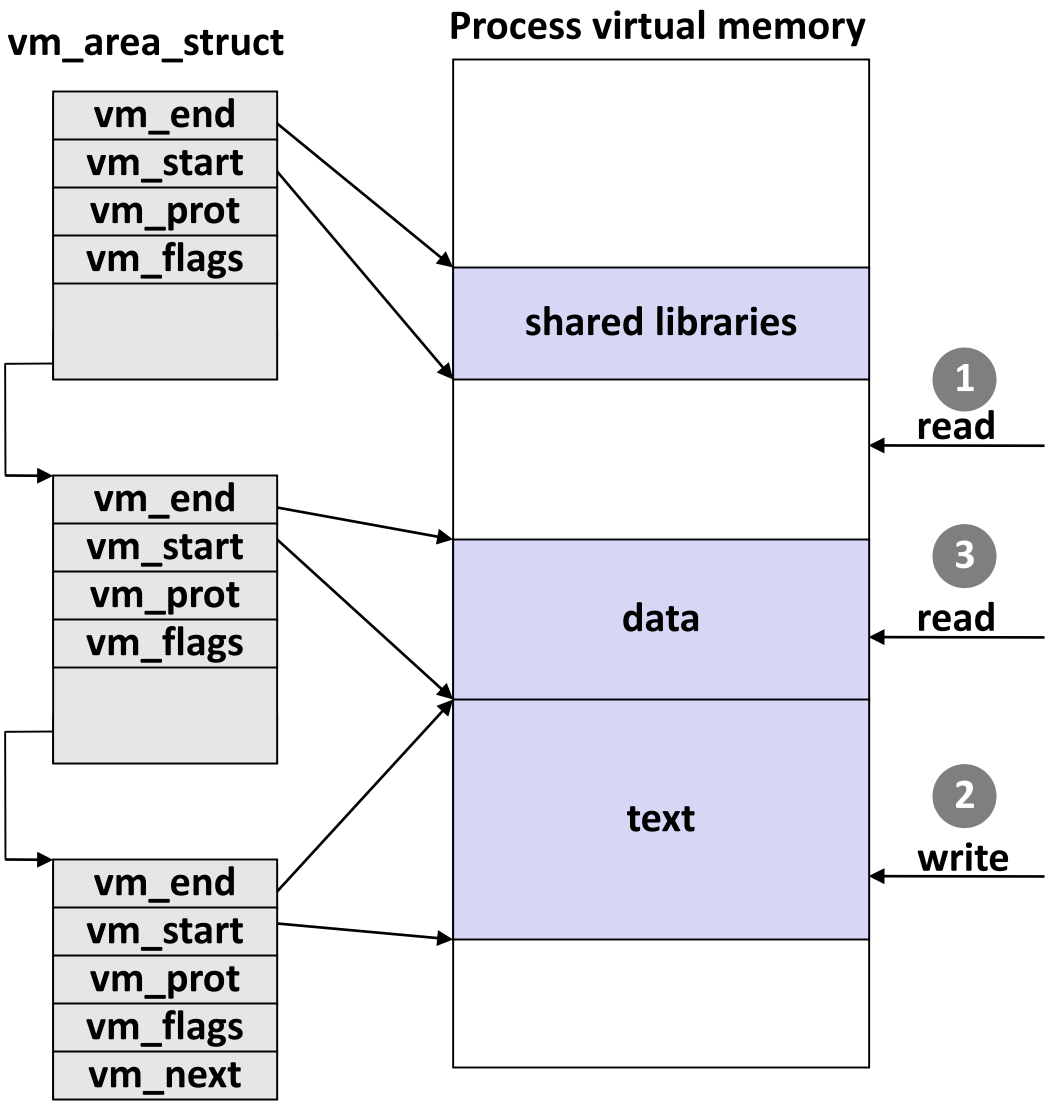
Linux Organizes VM as Collection of “Areas”



- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process

Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

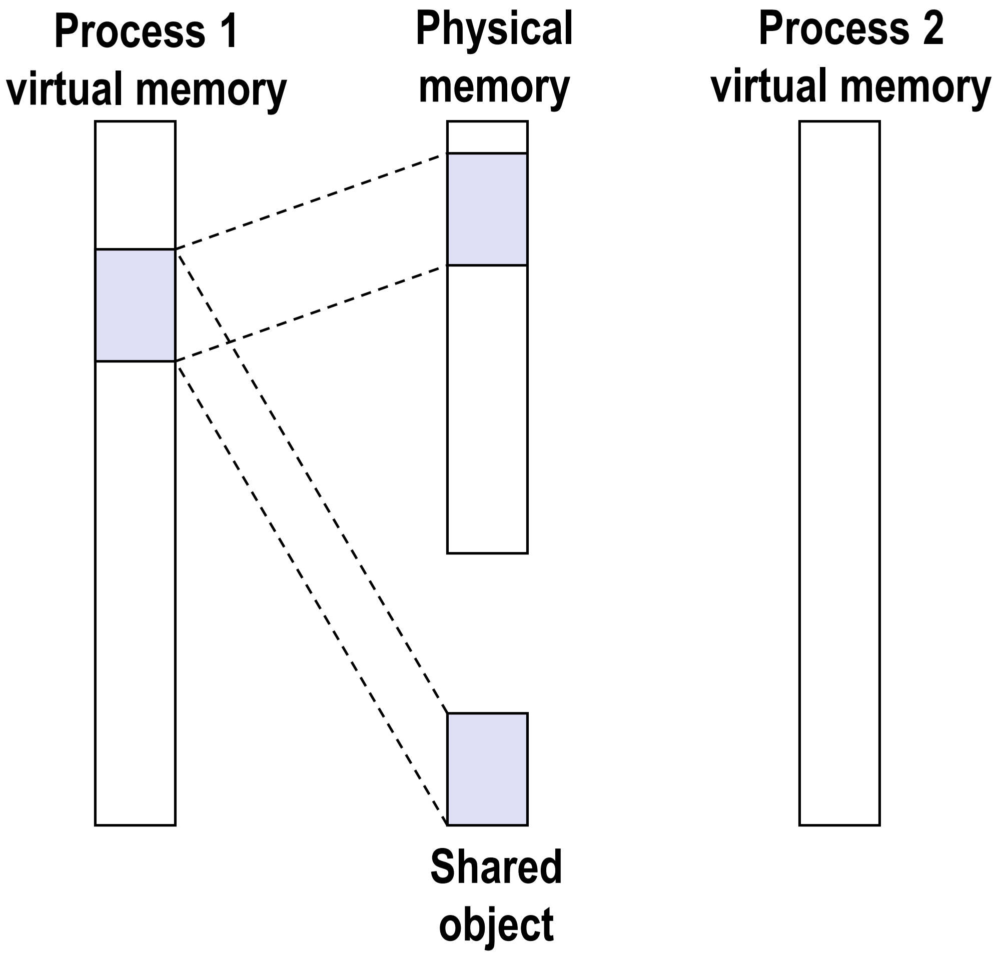
Today

- Simple memory system example
- Case study: Core i7/Linux memory system
- **Memory mapping**

Memory Mapping

- **VM areas initialized by associating them with disk objects.**
  - Process is known as ***memory mapping***.
- **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
  - ***Anonymous file*** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (***demand-zero page***)
    - Once the page is written to (***dirtied***), it is like any other page
- **Dirty pages are copied back and forth between memory and a special *swap file*.**

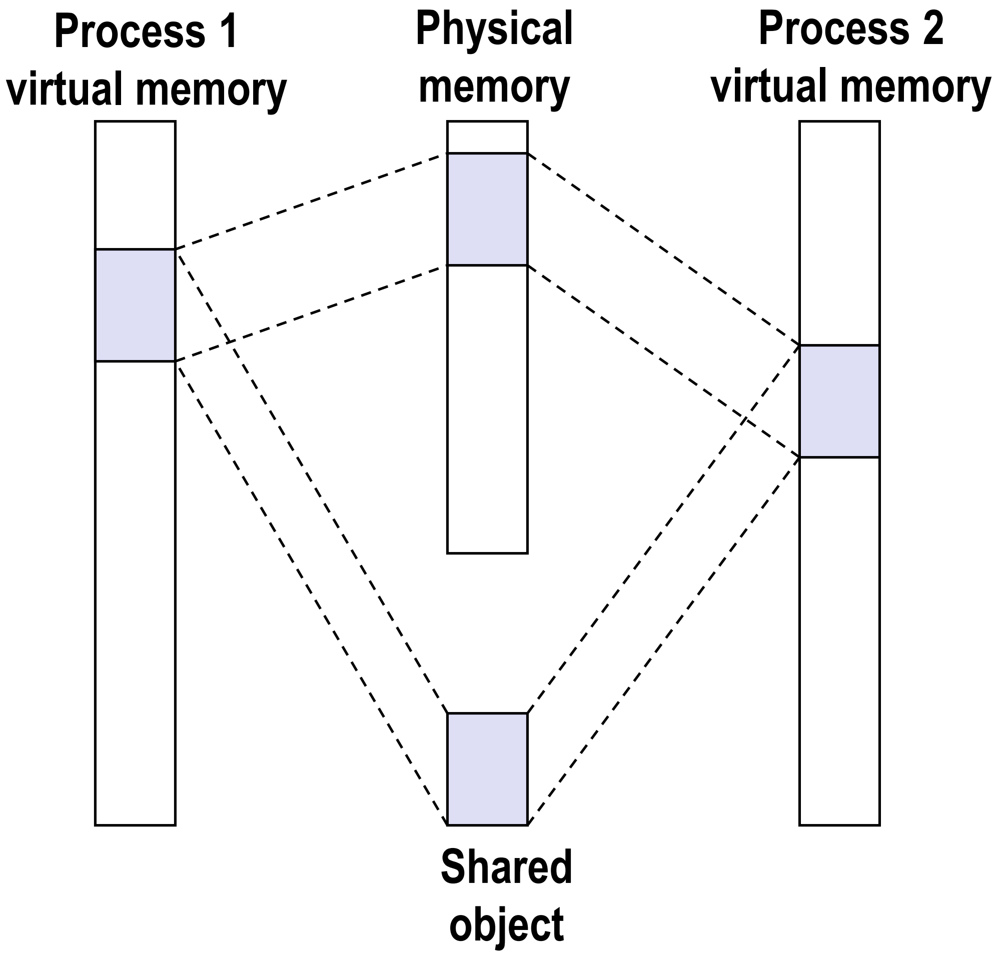
Sharing Revisited: Shared Objects



- Process 1 maps the shared object.

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

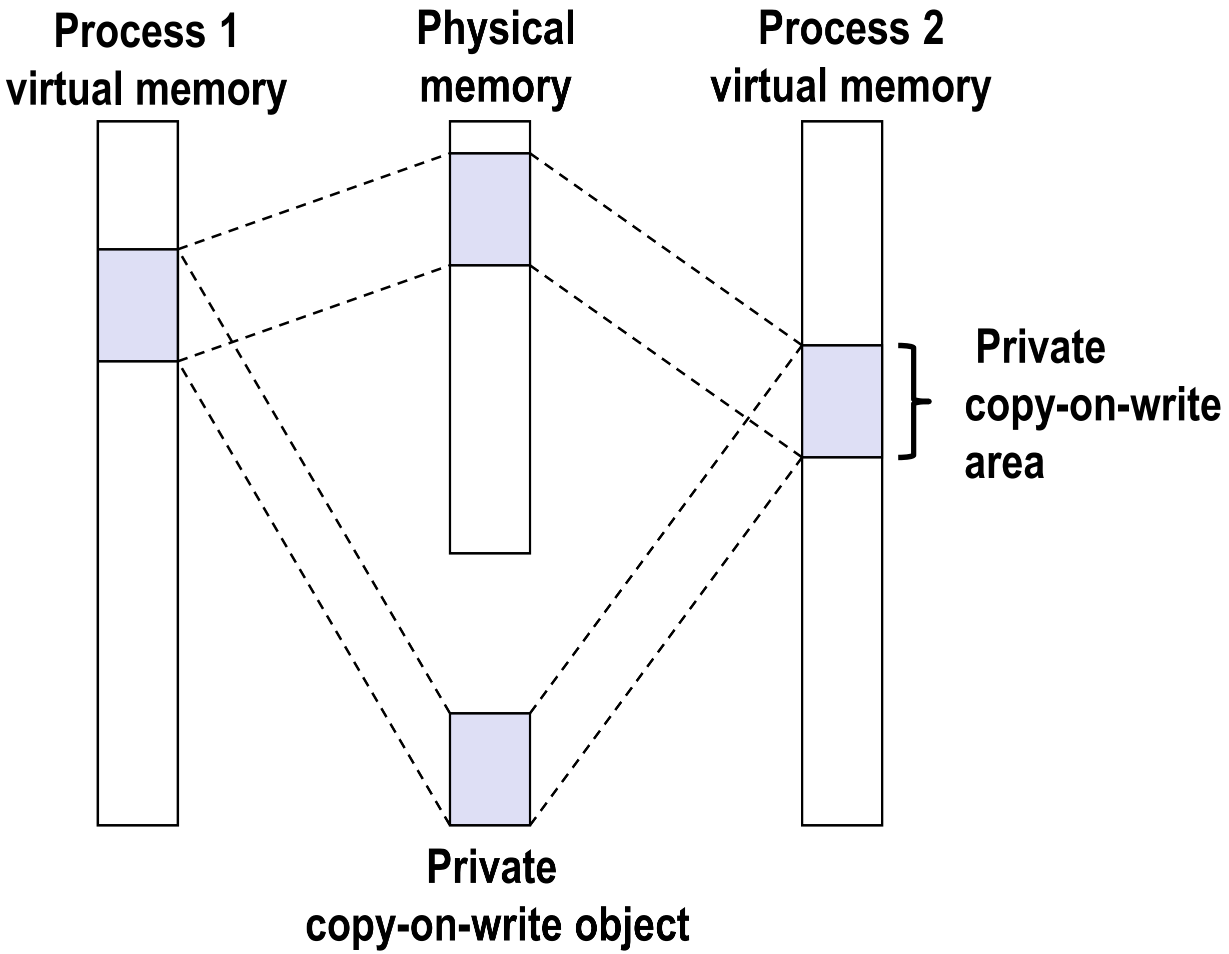
Sharing Revisited: Shared Objects



- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

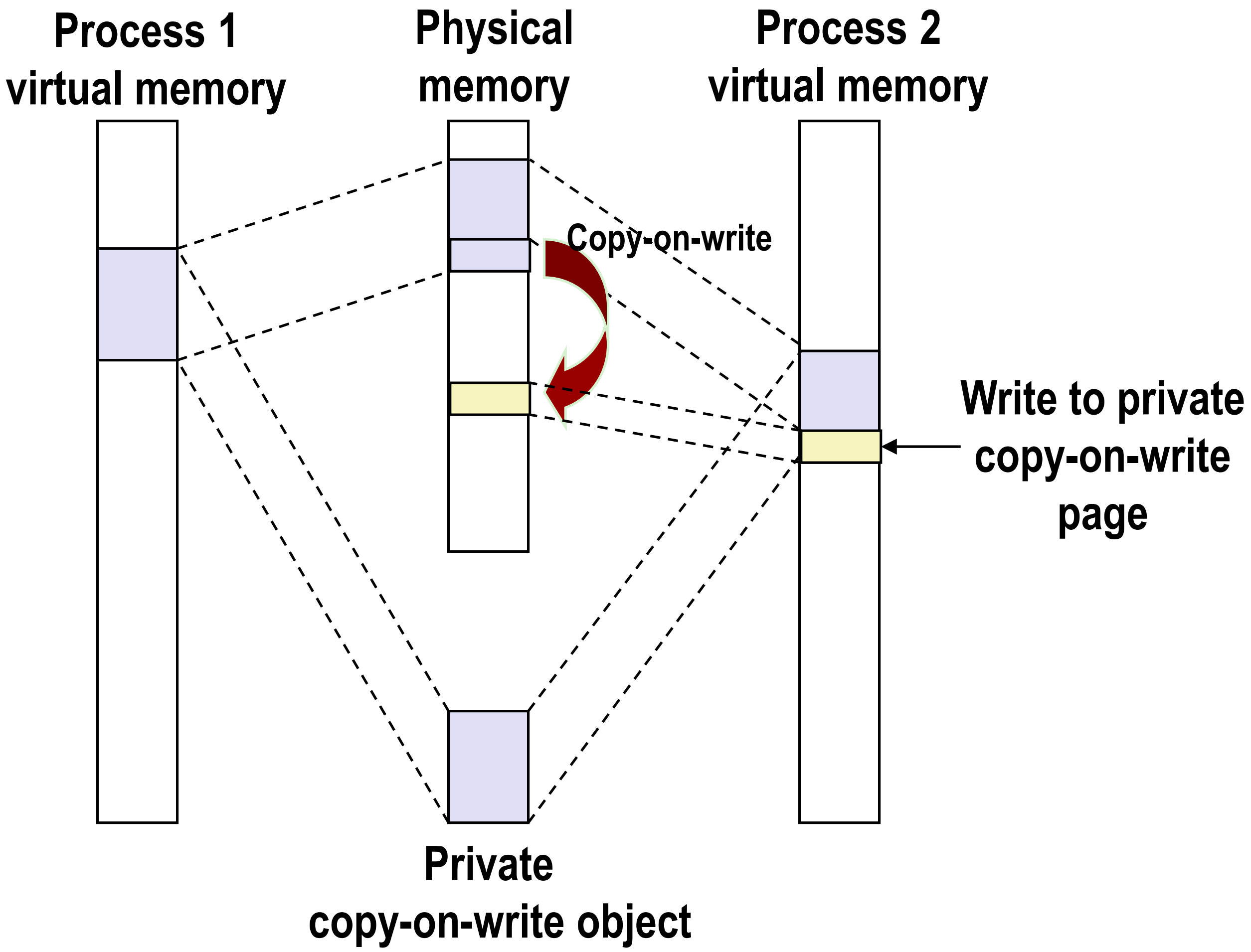
Sharing Revisited: Private Copy-on-write (COW) Objects



- Two processes mapping a ***private copy-on-write (COW)*** object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Sharing Revisited:
Private Copy-on-write (COW) Objects

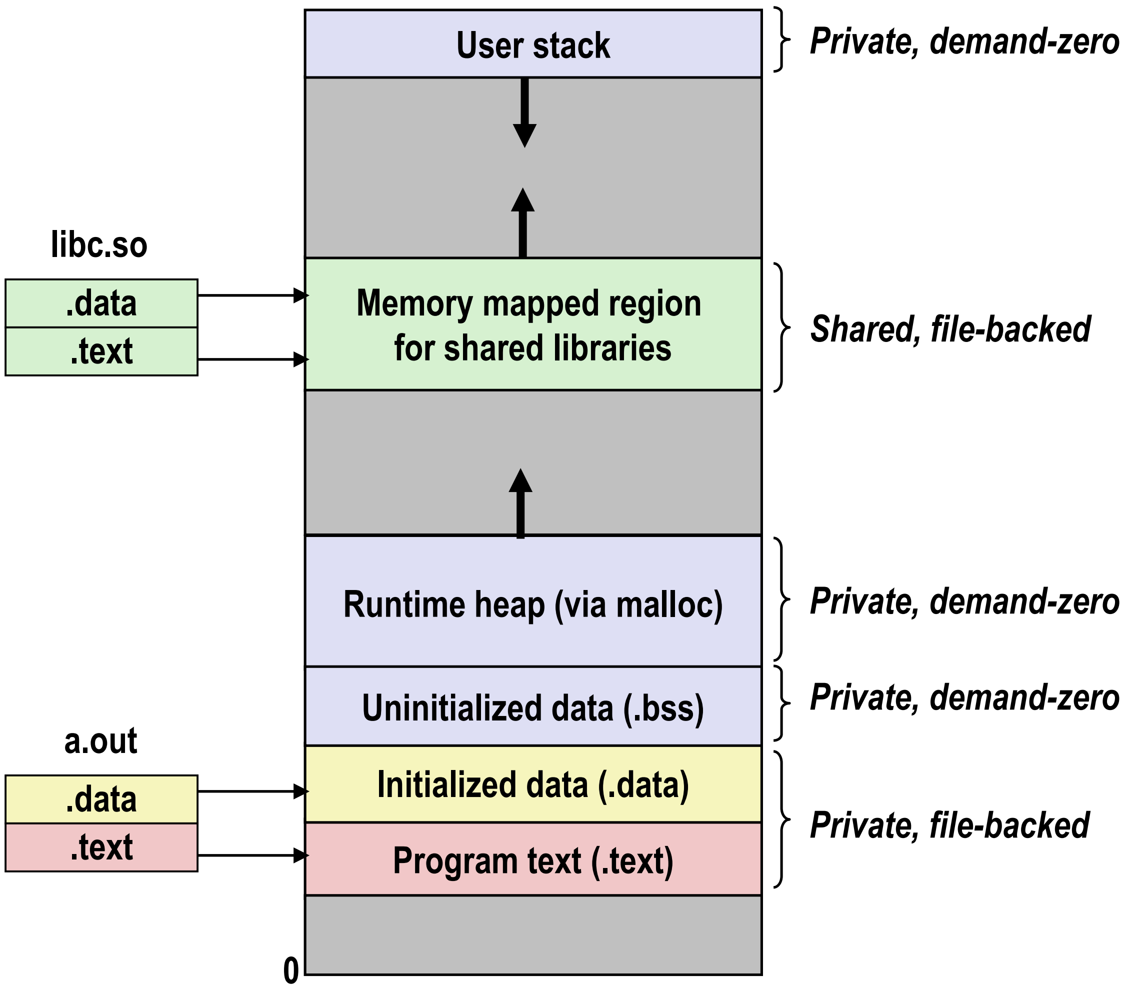


- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

The `fork` Function Revisited

- **VM and memory mapping explain how `fork` provides private address space for each process.**
- **To create virtual address for new new process**
  - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW
- **On return, each process has exact copy of virtual memory**
- **Subsequent writes create new pages using COW mechanism.**

The `execve` Function Revisited



- **To load and run a new program `a.out` in the current process using `execve`:**
- **Free `vm_area_struct`'s and page tables for old areas**
- **Create `vm_area_struct`'s and page tables for new areas**
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files .
- **Set PC to entry point in `.text`**
  - Linux will fault in code and data pages as needed.

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

User-Level Memory Mapping

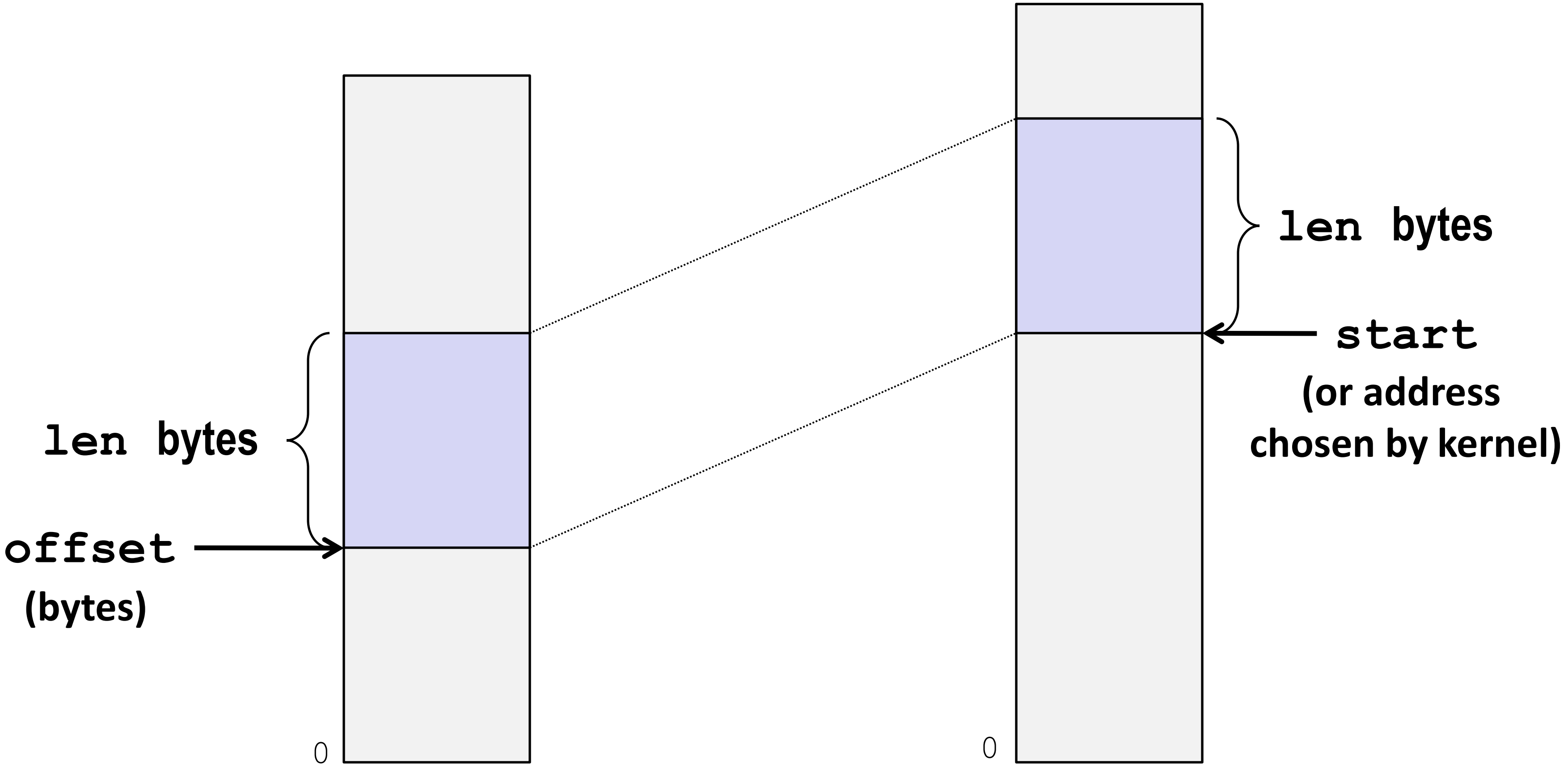
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- **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`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- **Return a pointer to start of mapped area (may not be `start`)**

User-Level Memory Mapping

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



Disk file specified by file descriptor `fd`

Process virtual memory

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Example: Using `mmap` to Copy Files

- **Copying a file to `stdout` without transferring data to user space .**

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

   /* Ptr to memory mapped area */
   char *bufp;

   bufp = Mmap(NULL, size,
            PROT_READ,
            MAP_PRIVATE,
            fd, 0);
   Write(1, bufp, size);
   return;
}
```

mmapcopy.c

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
   struct stat stat;
   int fd;

   /* Check for required cmd line arg */
   if (argc != 2) {
      printf("usage: %s <filename>\n",
           argv[0]);
      exit(0);
   }

   /* Copy input file to stdout */
   fd = Open(argv[1], O_RDONLY, 0);
   Fstat(fd, &stat);
   mmapcopy(fd, stat.st_size);
   exit(0);
}
```

mmapcopy.c

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition