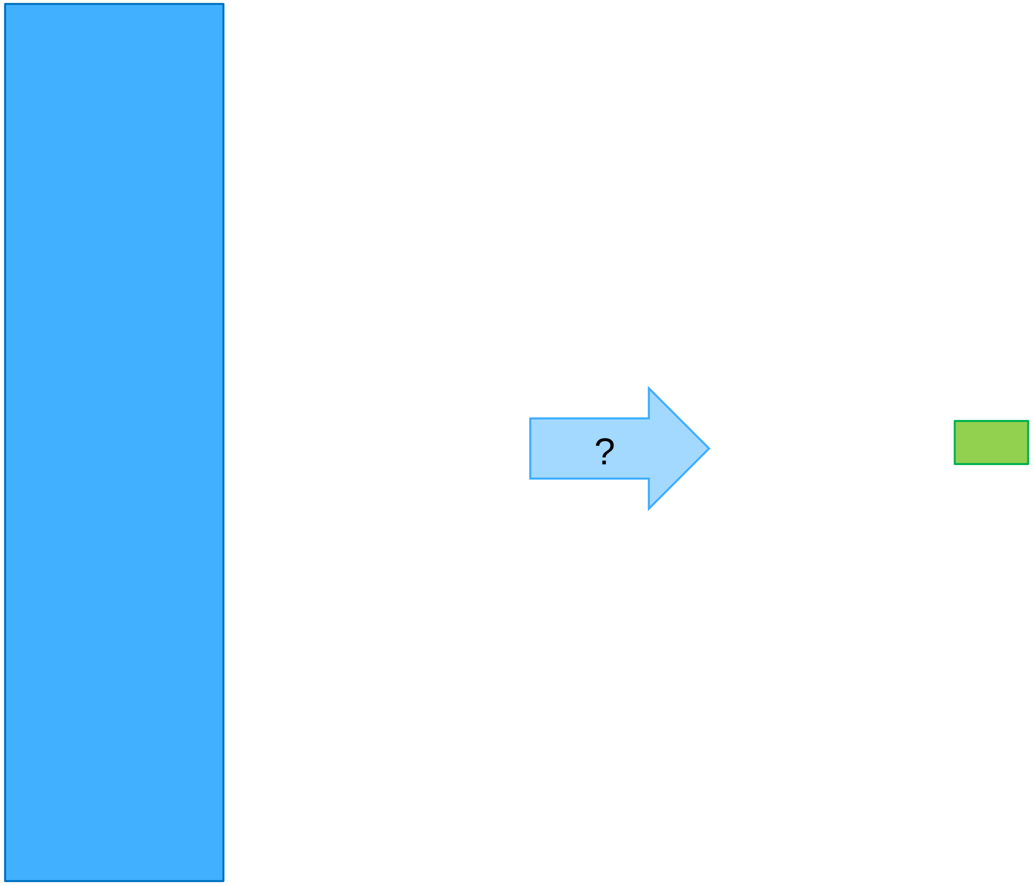
Running Programs on a System

# Virtual Memory

# Virtual Memory: Theory

# Problem 1: How does everything fit?

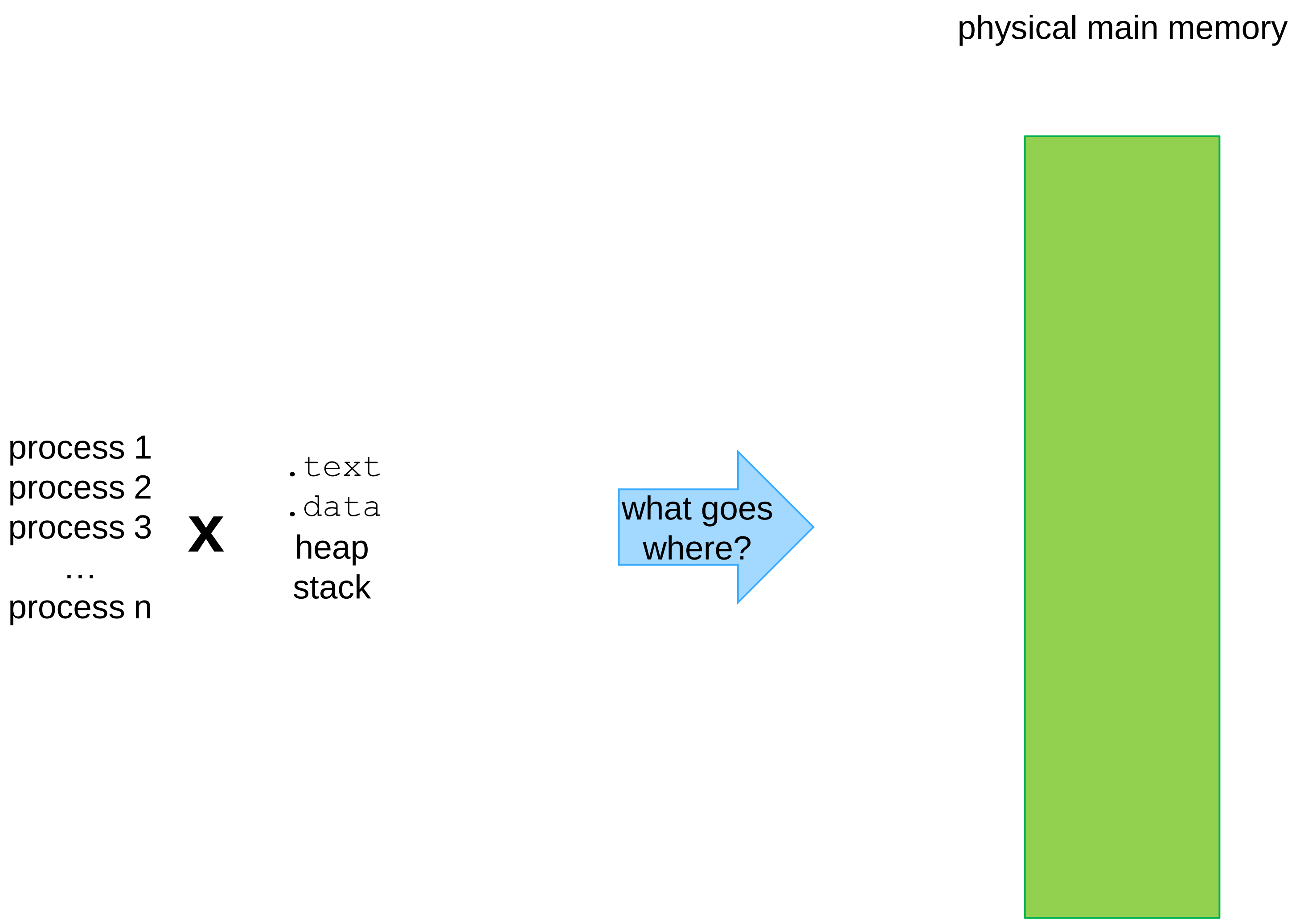


virtual memory:
64-bit addresses,
16 exabytes

physical memory:
a few gigabytes

…and there are many processes

# Problem 2: Memory Management



physical main memory

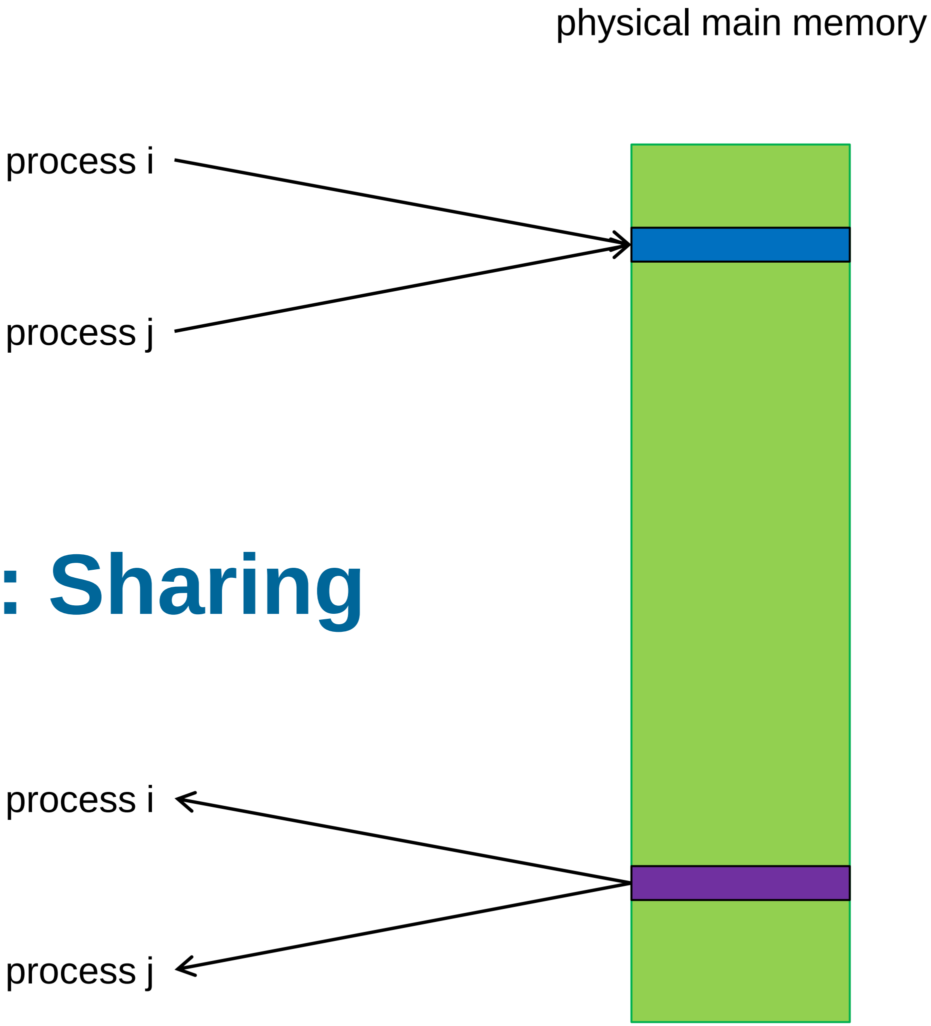
process 1
process 2
process 3
…
process n

x

`.text`
`.data`
heap
stack

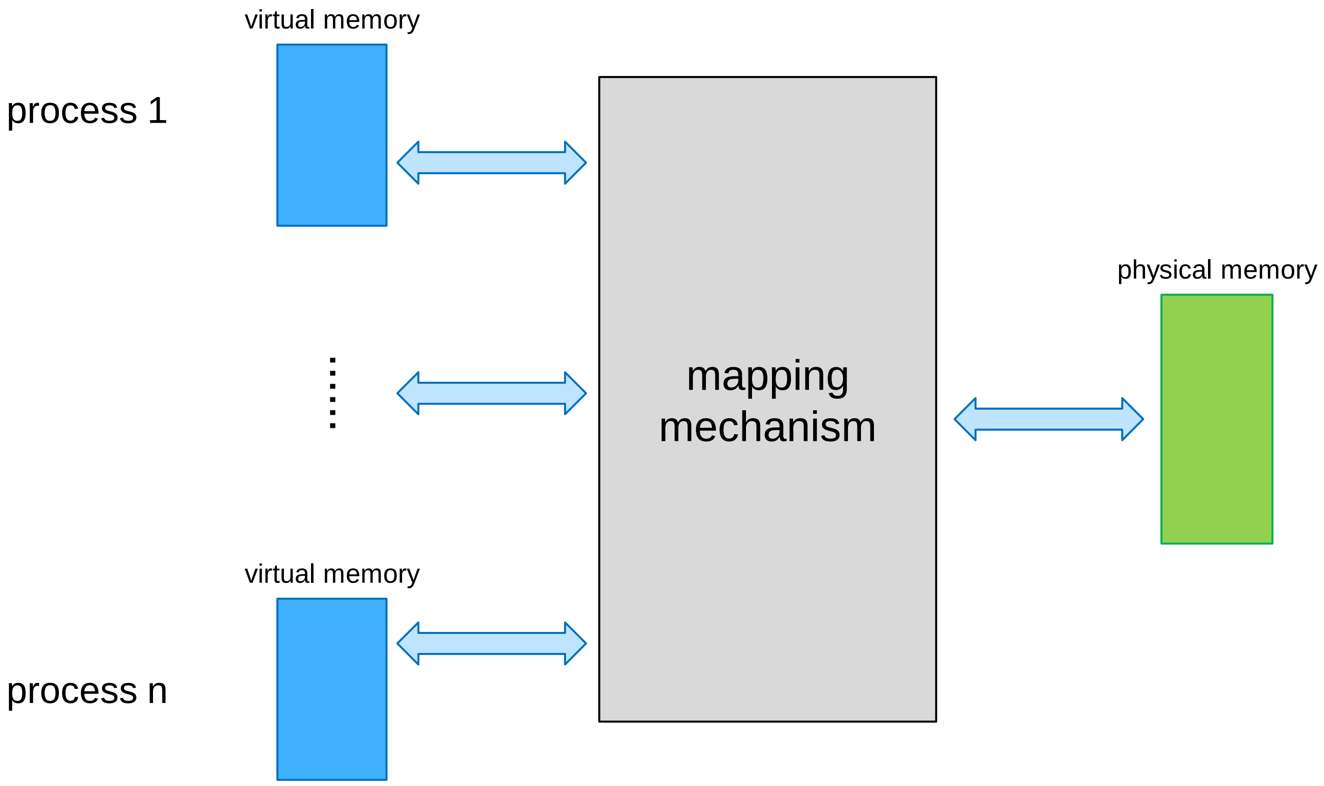
what goes where?

# Problem 3: Protection



# Problem 4: Sharing

# Solution: Indirection



- Each process gets its own private memory space
- Solves all previous problems

# Address Spaces

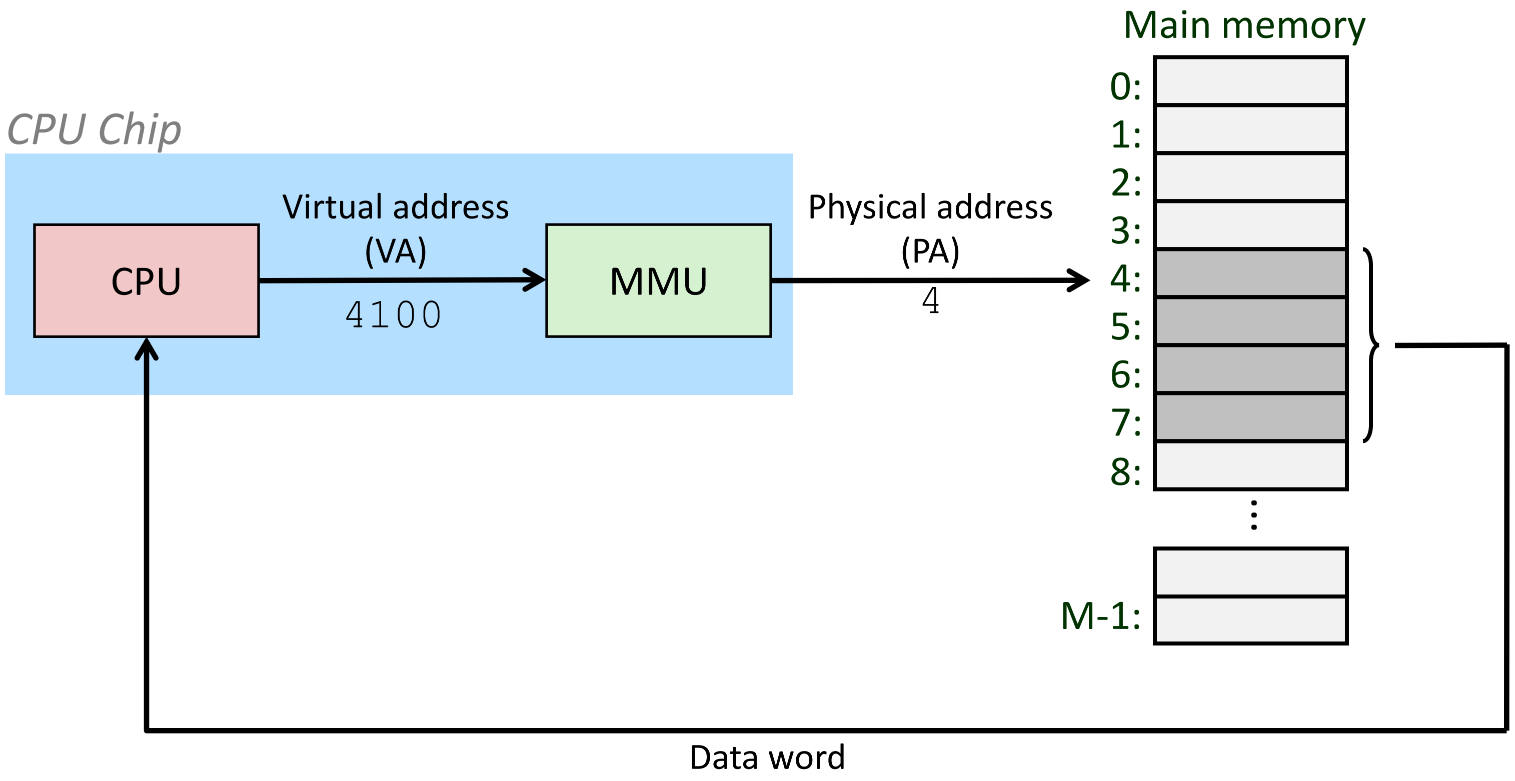
- Linear address space: Ordered set of contiguous non-negative integer addresses:
  {0, 1, 2, 3 … }

- 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}

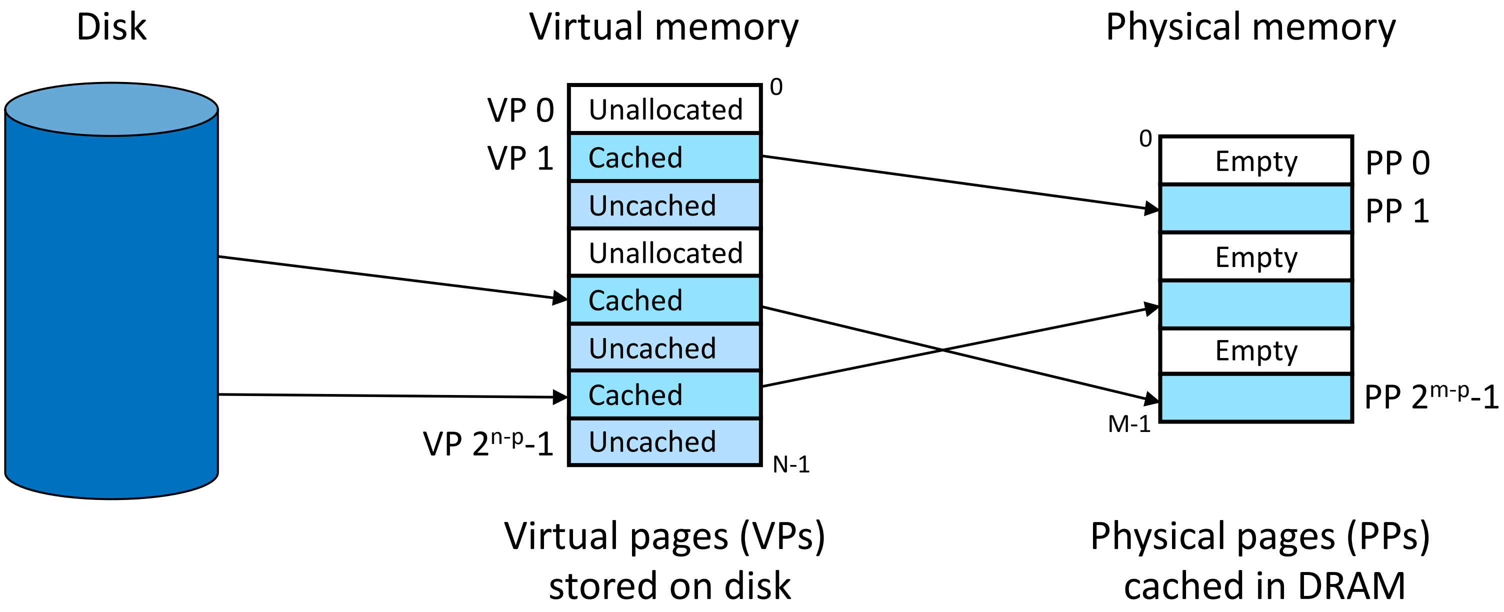
- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses

# Virtual Addressing



# VM as a Tool for Caching

- Virtual memory is an array of N contiguous bytes stored on disk
  - conceptually: stored somewhere on disk
- Physical memory = cache for allocated virtual memory
- Cache blocks are called pages (size is P = $2^p$ bytes)



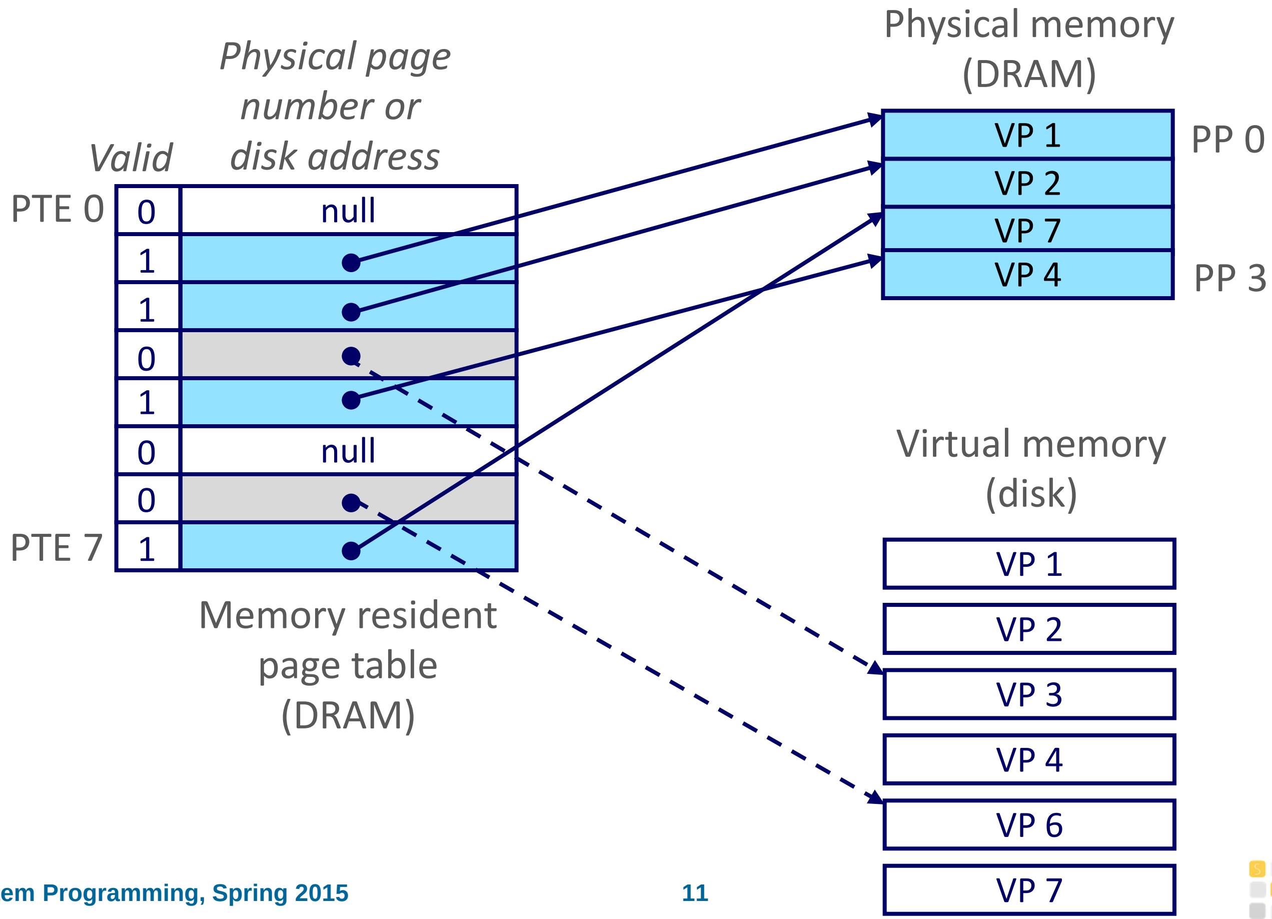
# DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM

- Consequences
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

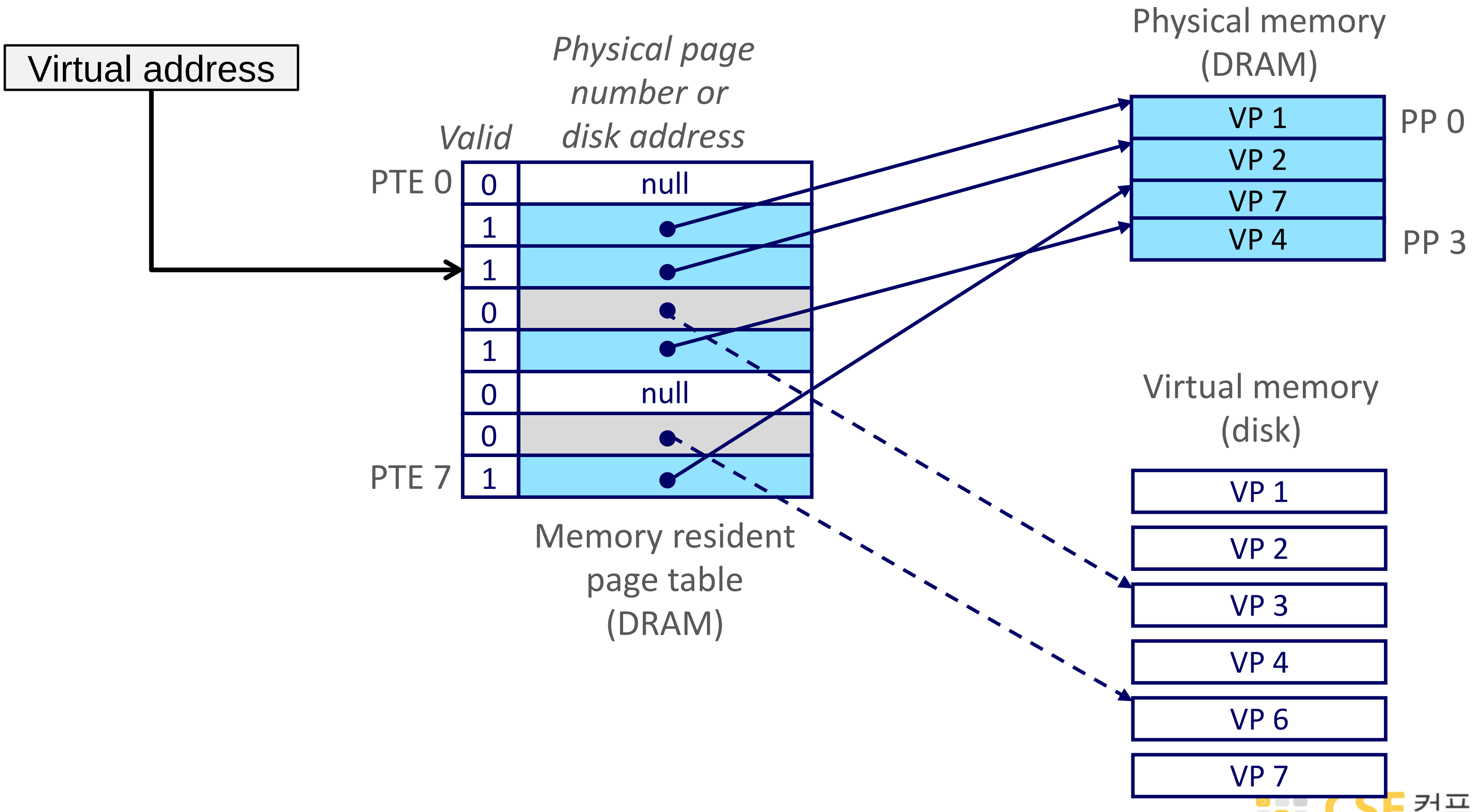
# Address Translation: Page Tables

- 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



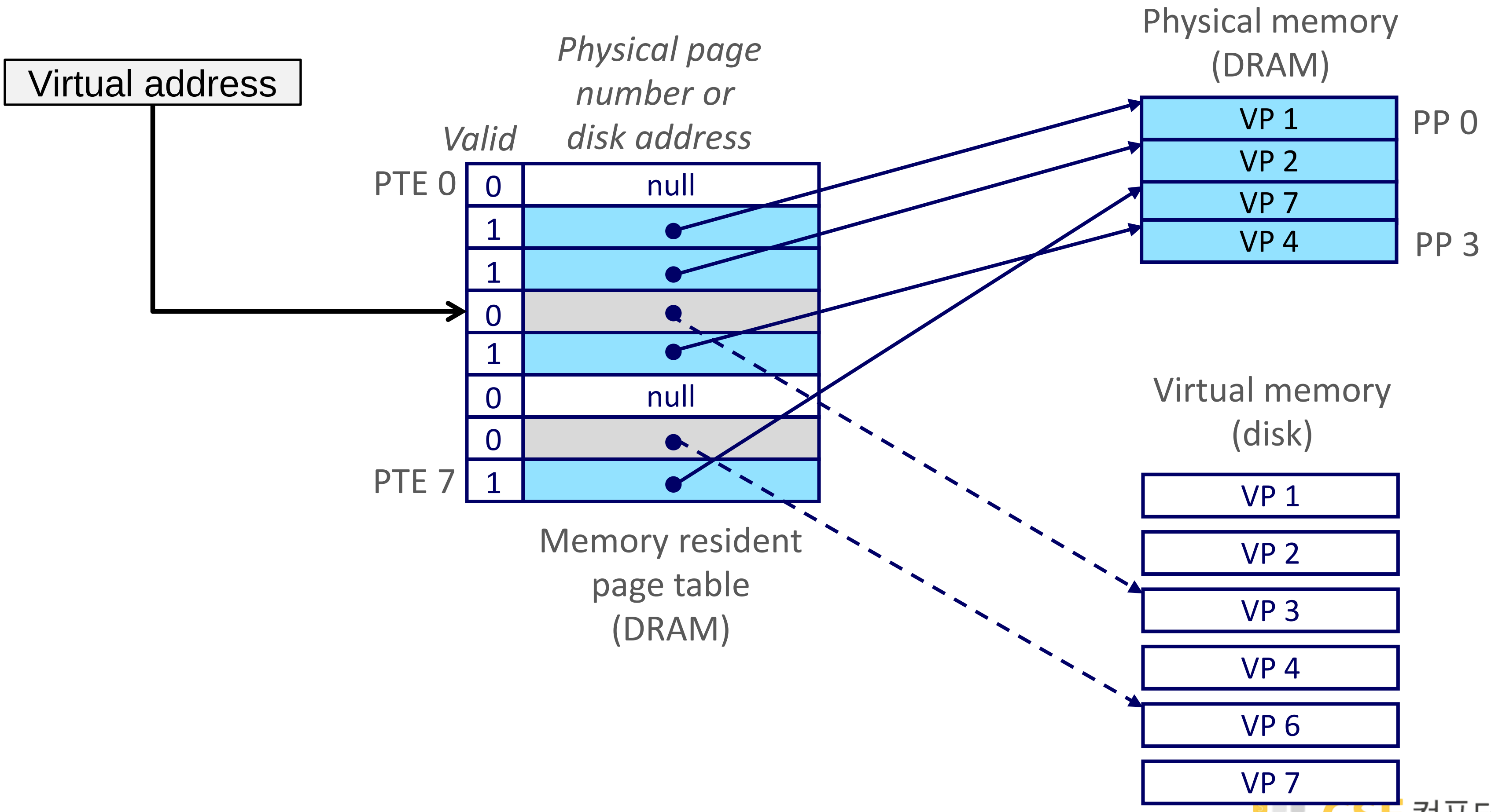
# Page Hit

- Reference to VM word that is in physical memory (DRAM cache hit)



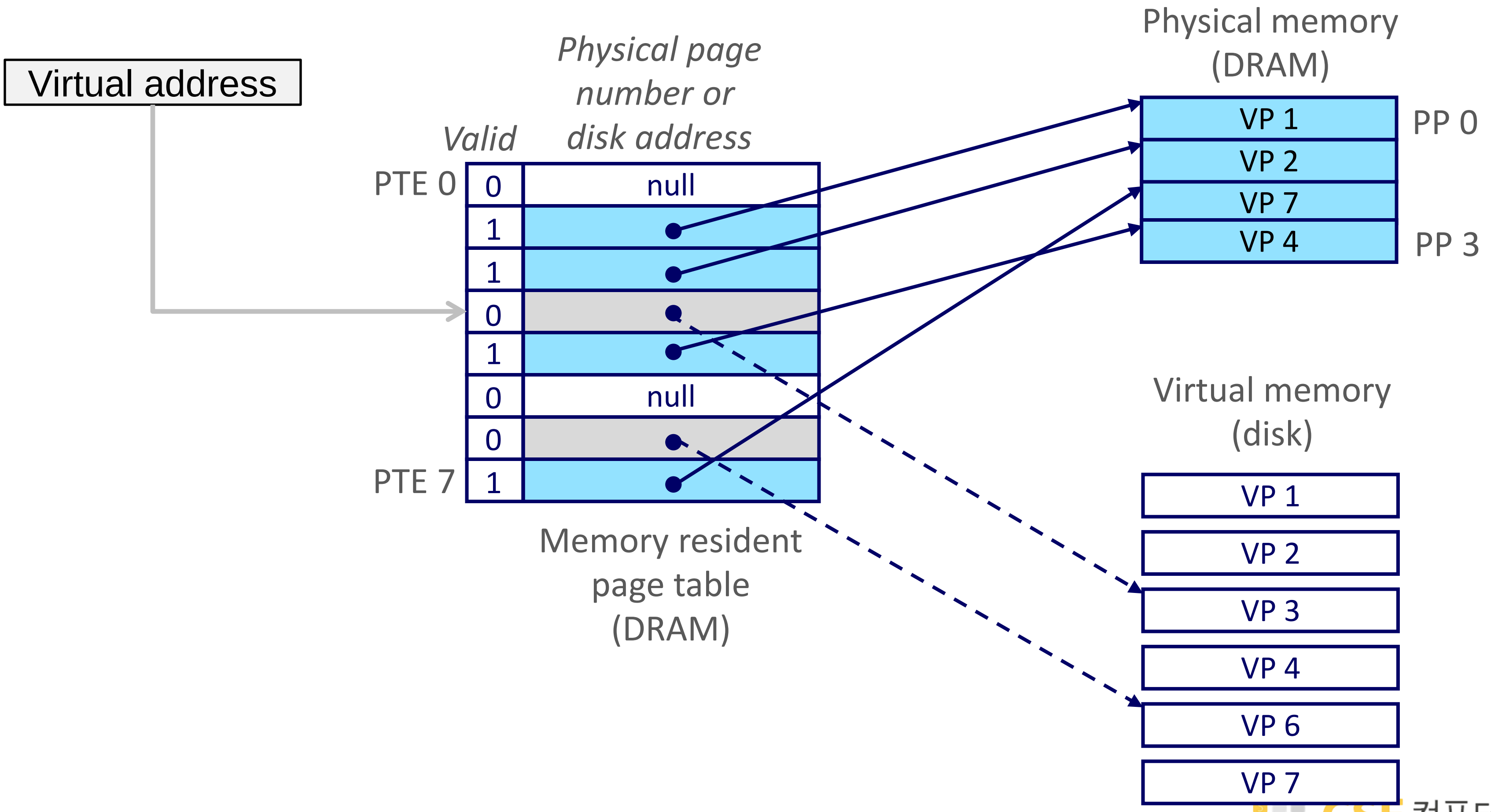
# Page Miss

- Reference to VM word that is not in physical memory
  - in the context of VM called "page fault"



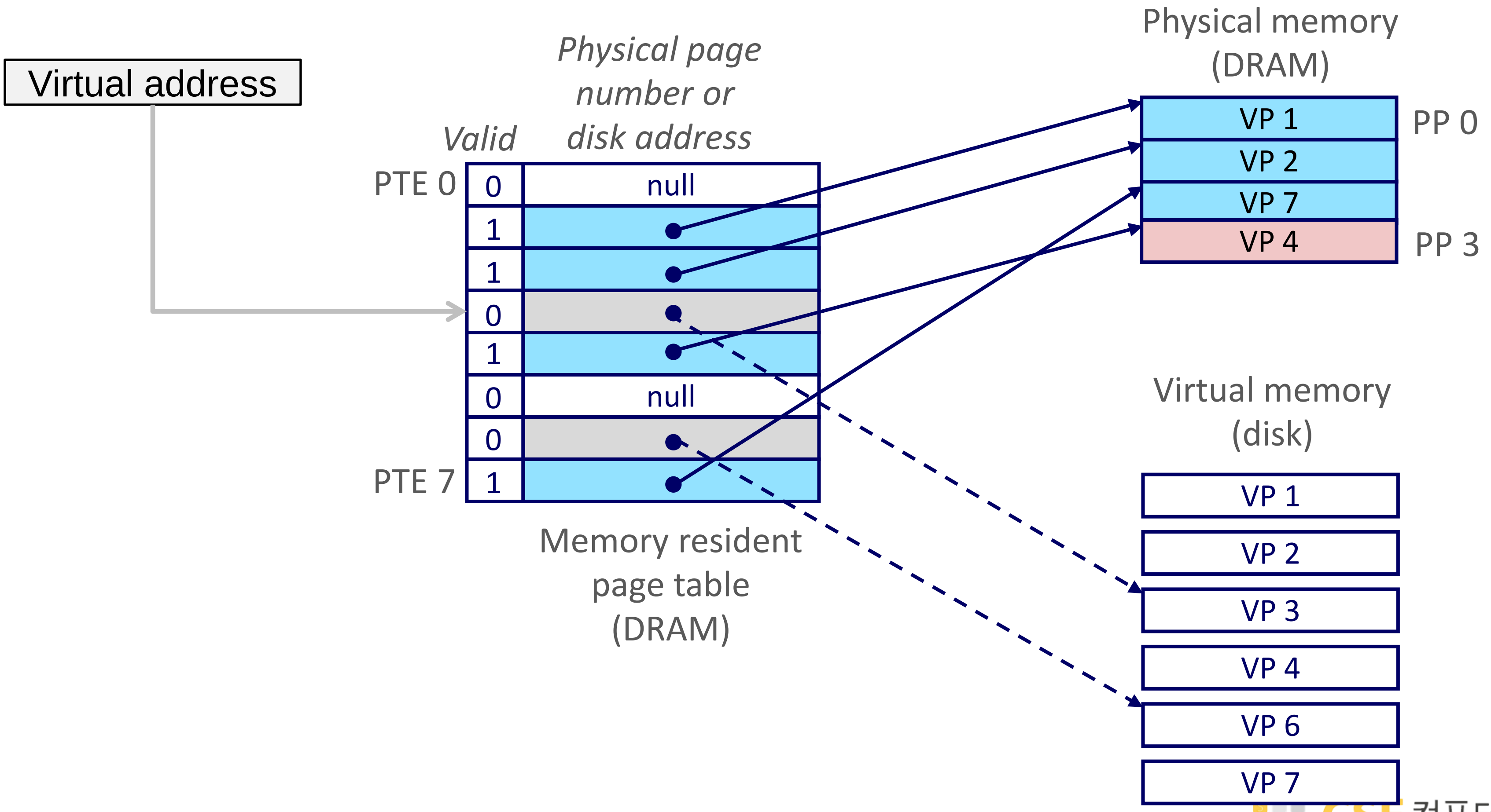
# Handling a Page Fault

- Page miss causes page fault (an exception)



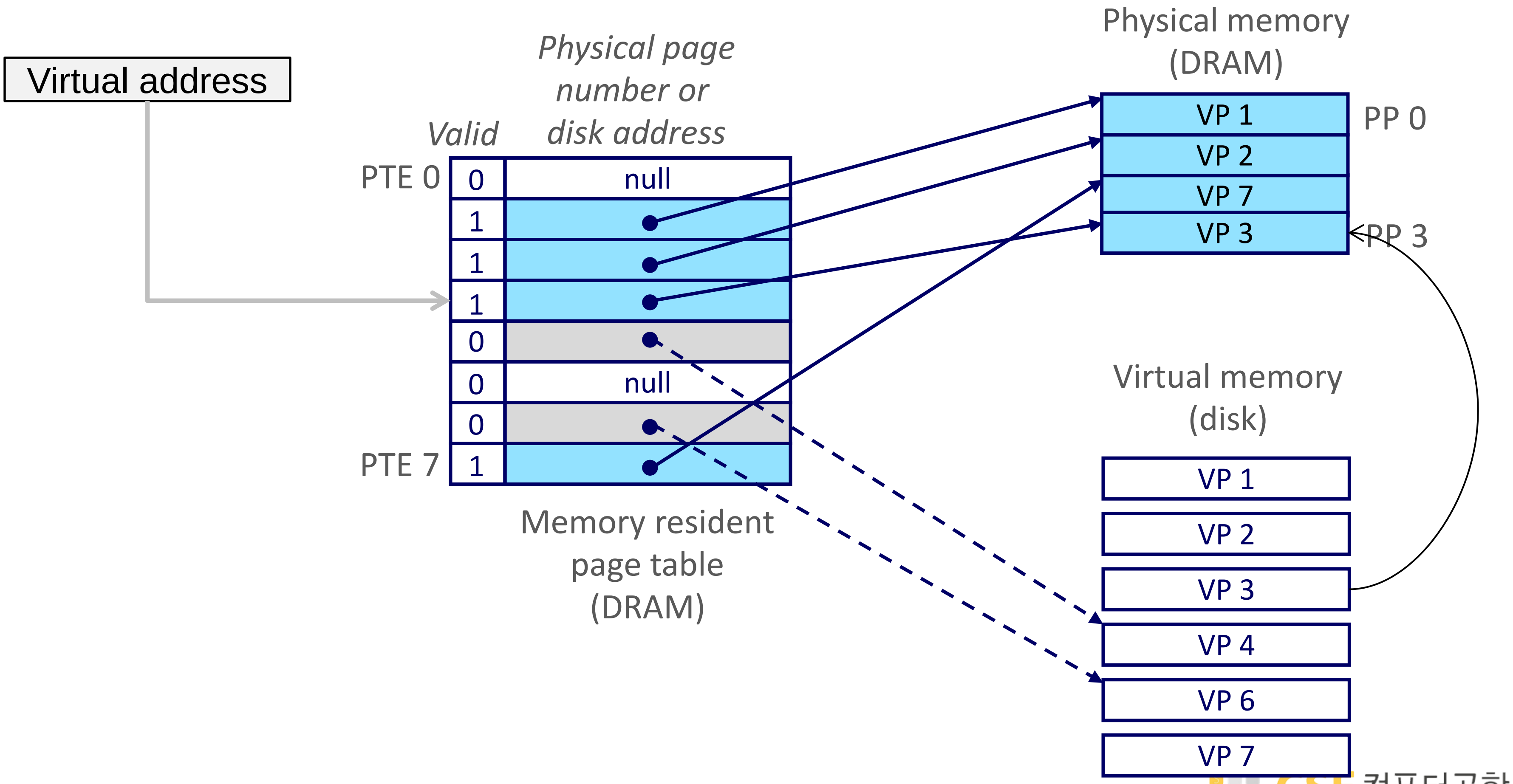
# Handling a Page Fault

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



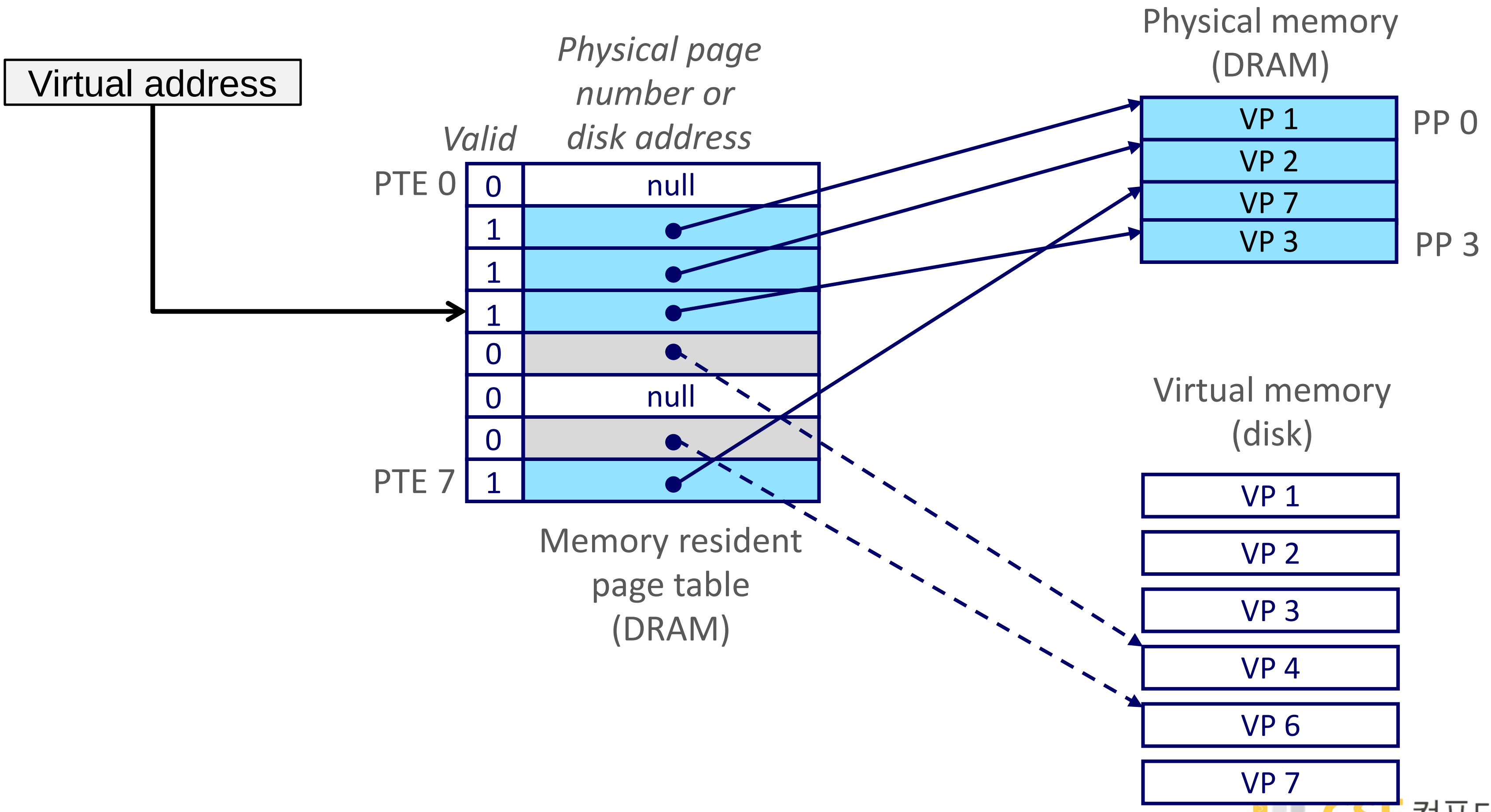
# Handling a Page Fault

- Page fault handler selects a victim to be evicted (here VP 4) and loads the requested page into physical memory



# Handling Page Fault

- Offending instruction is restarted: page hit!



# Why does it work? Locality

- Virtual memory works because of locality

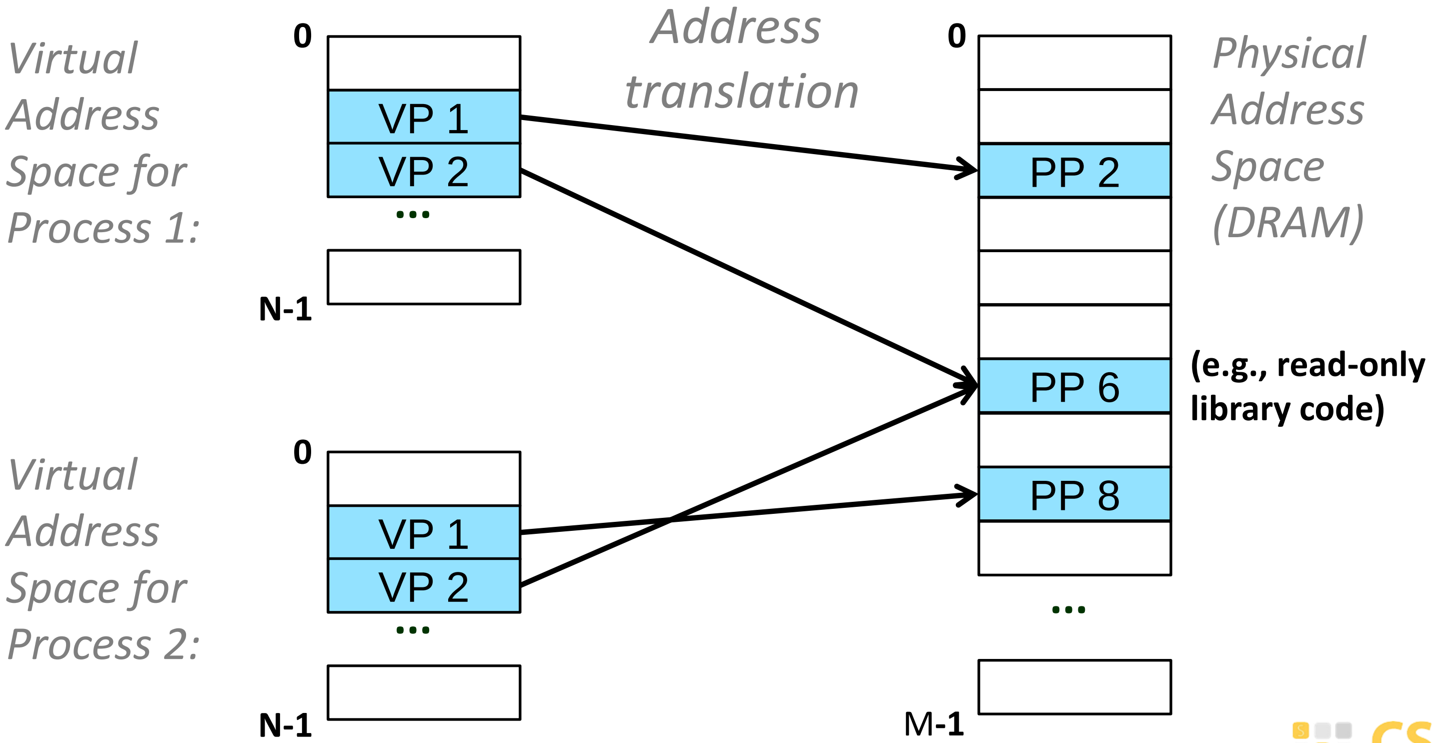
- At any point in time, programs tend to access a set of active virtual pages called the **working set**
  - Programs with better temporal locality have smaller working sets

- 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

# VM as a Tool for Memory Management

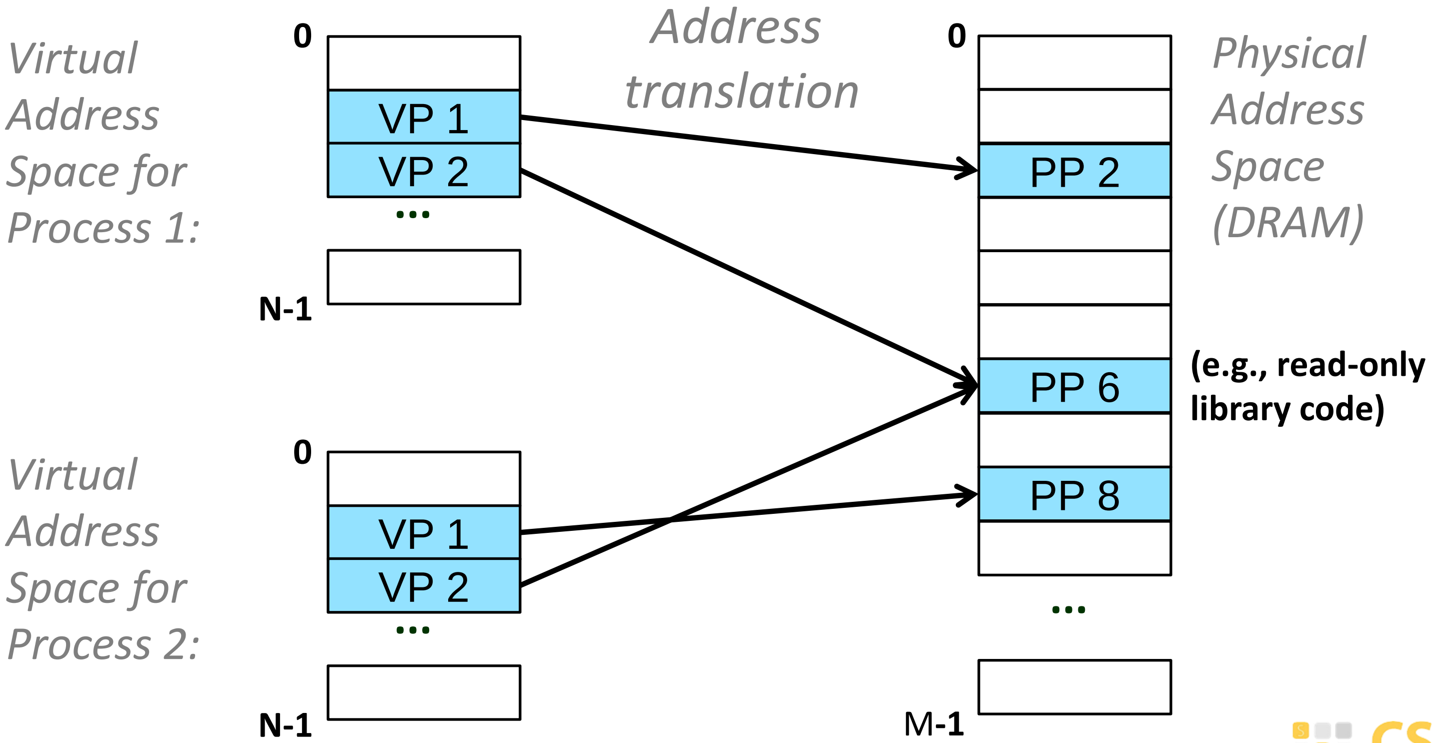
- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well chosen mappings simplify memory allocation and management


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)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

(e.g., read-only library code)

# VM as a Tool for Memory Management

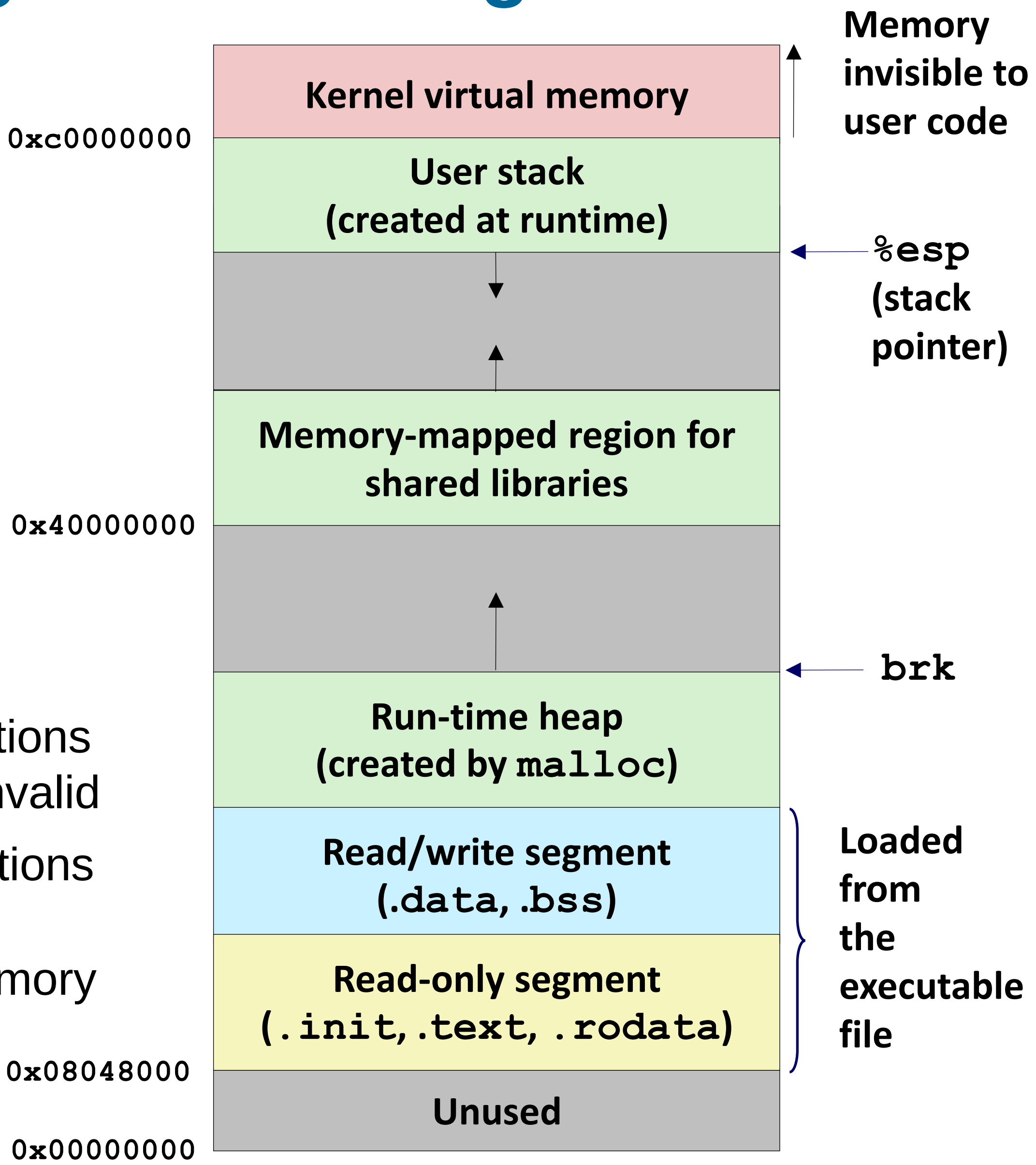
- Memory allocation
  - Each virtual page can be mapped to *any* physical page
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)



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

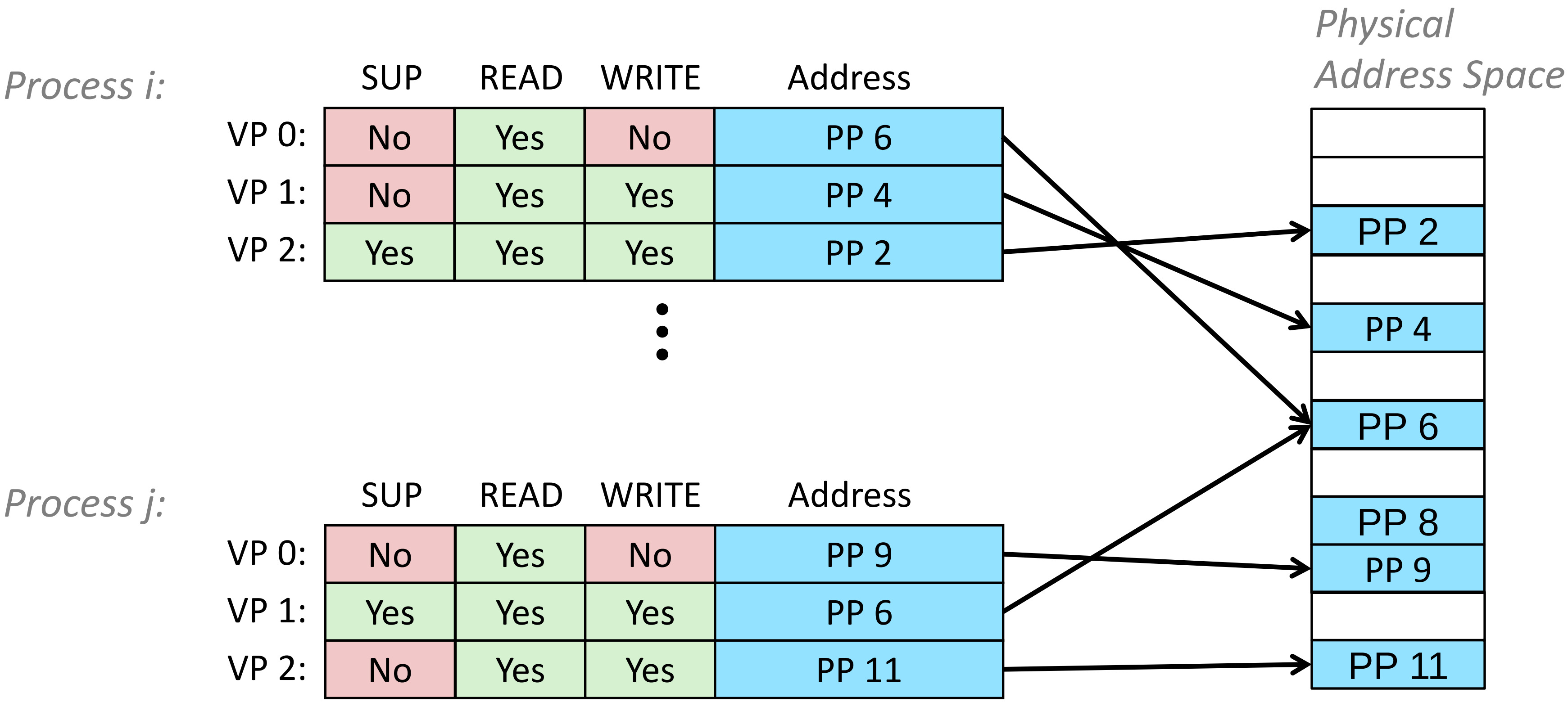
# Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address
- Loading
  - `execve()` 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
0xc0000000
User stack (created at runtime)
%esp (stack pointer)
Memory-mapped region for shared libraries
0x40000000
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Loaded from the executable file
Read-only segment (.init, .text, .rodata)
0x08048000
Unused
0x00000000

# 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)



*Process i:*

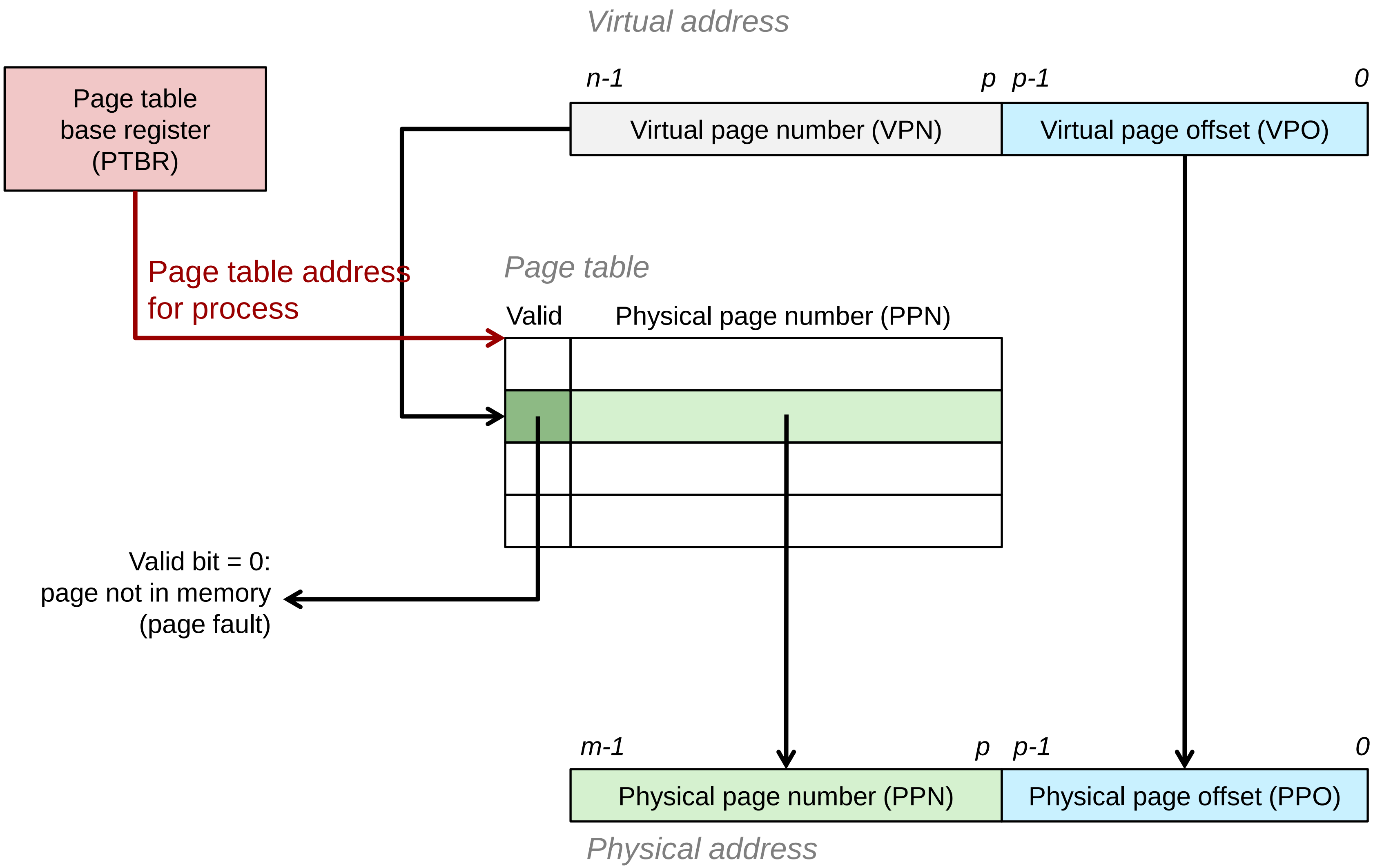
|  | SUP | READ | WRITE | Address |
|---|---|---|---|---|
| VP 0: | No | Yes | No | PP 6 |
| VP 1: | No | Yes | Yes | PP 4 |
| VP 2: | Yes | Yes | Yes | PP 2 |

⋮

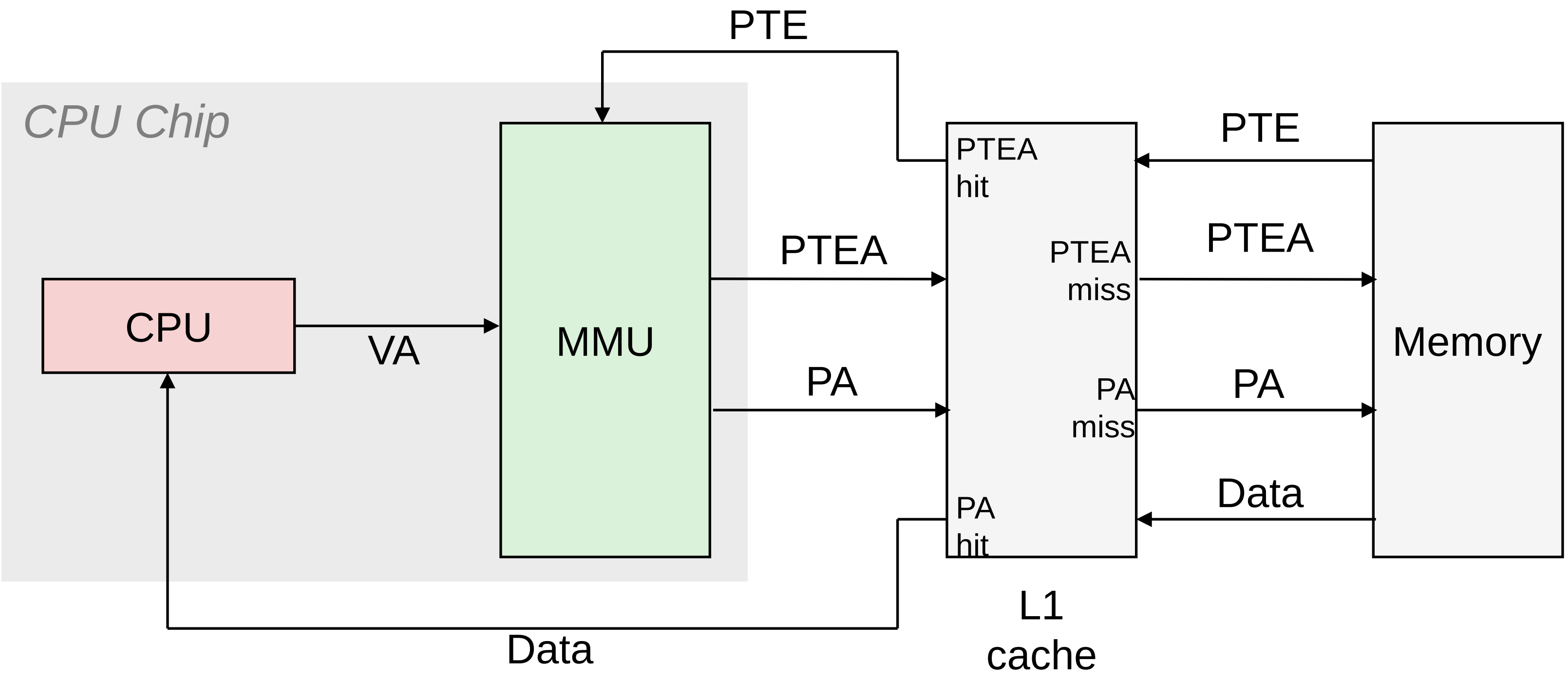
*Process j:*

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

# Address Translation


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

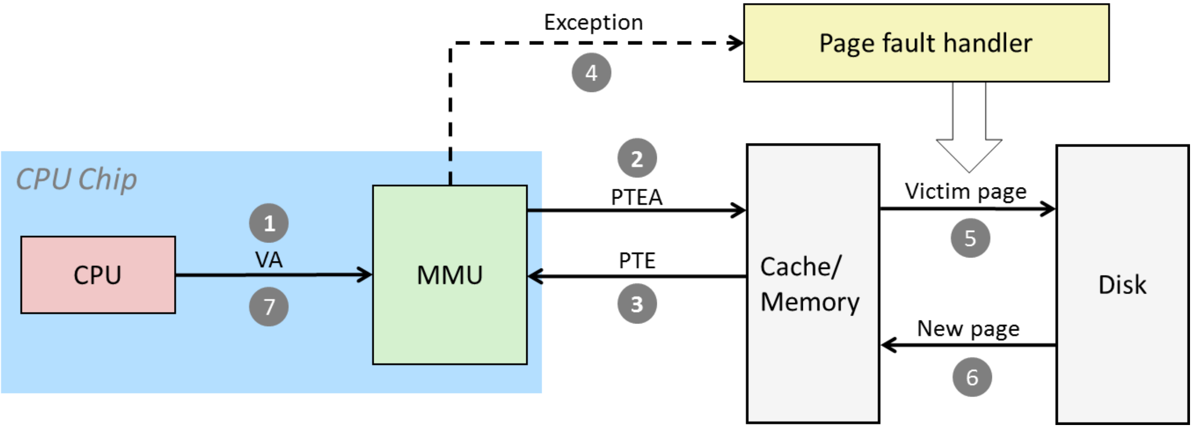
# Integrating VM and Cache



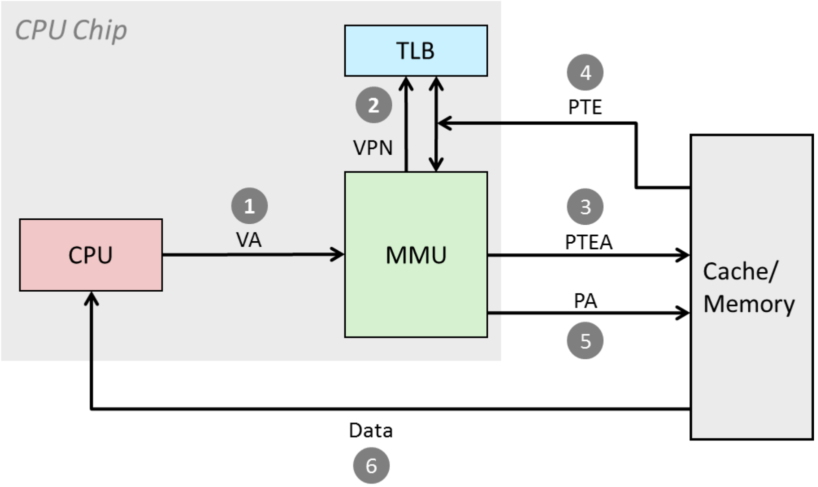
*VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address*

# Page Hit/Fault, TLB Hit/Miss

- Page Hit/Fault
  - hit: direct access
  - miss:
    - load page from secondary storage
    - modify PTE
    - restart instruction

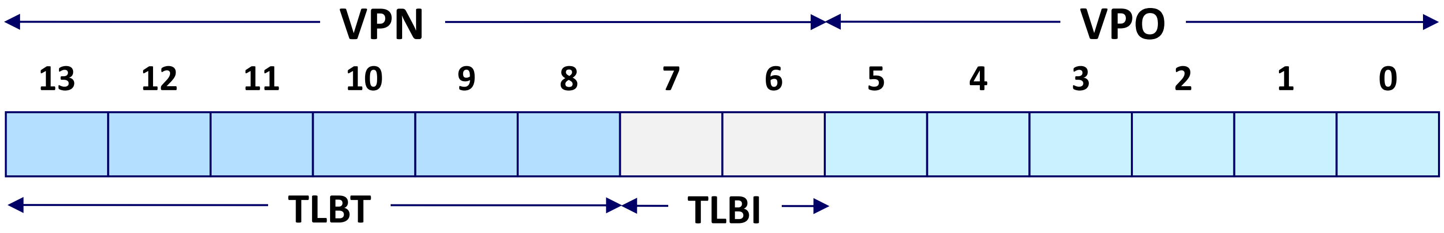


- TLB (Translation Lookaside Buffer) Hit/Miss
  - small cache in the MMU
  - eliminates one memory access per hit



# Virtual Addresses and TLB Index/Tag

- The TLB is a small cache with a high associativity
  - T = $2^t$ sets
  - TLBI (TLB Index) = *t* least significant bits of VPN
  - TLBT (TLB Tag) = remaining bits of VPN



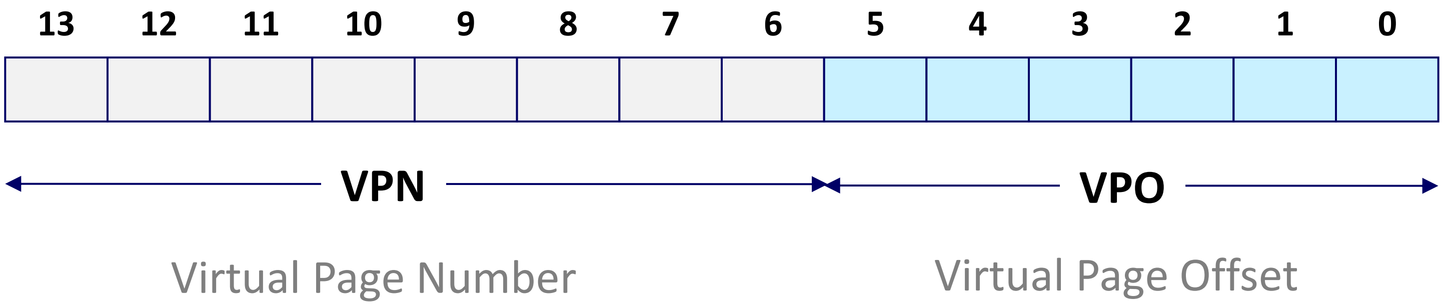
| Set | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 03 | – | 0 | 09 | 0D | 1 | 00 | – | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | – | 0 | 04 | – | 0 | 0A | – | 0 |
| 2 | 02 | – | 0 | 08 | – | 0 | 06 | – |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

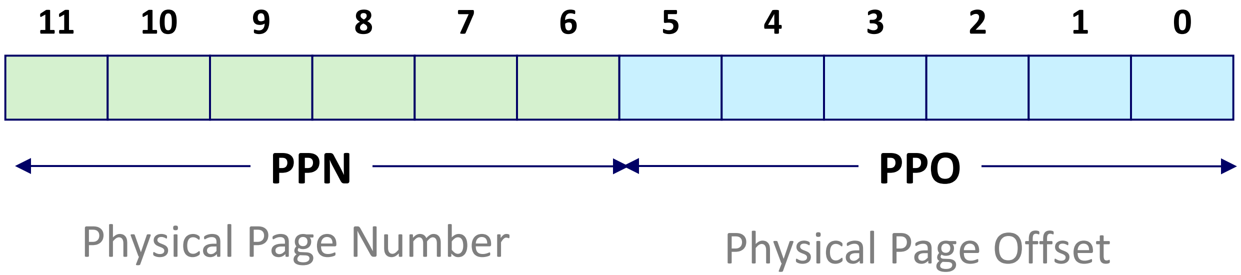
# Review of Symbols

- 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

# Simple Memory System Example

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





# Simple Memory System Page Table

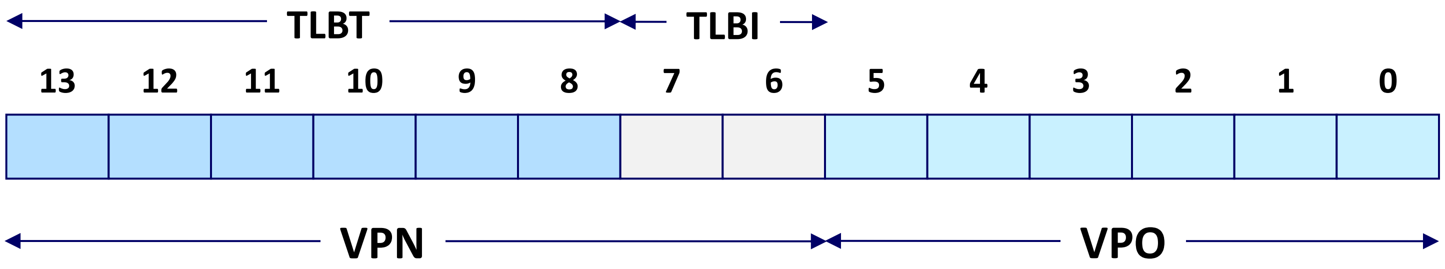
- Only show first 16 entries (out of 256)

| VPN | PPN | Valid |
|---|---|---|
| 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

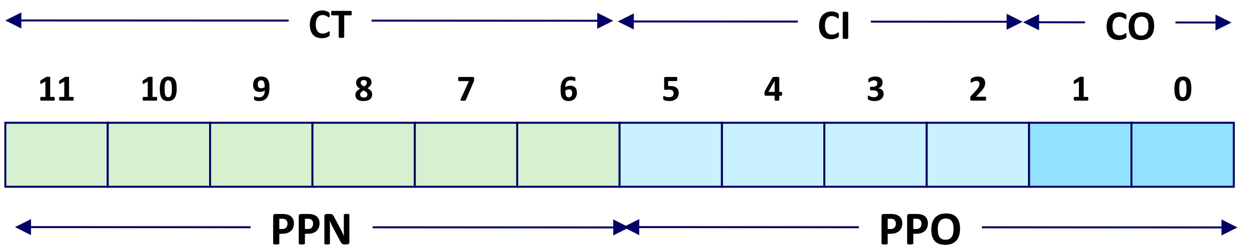
- 16 entries
- 4-way associative



| *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 |
| 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

| TLBT | | | | | | TLBI | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | | | | | | | | VPO | | | | | |

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

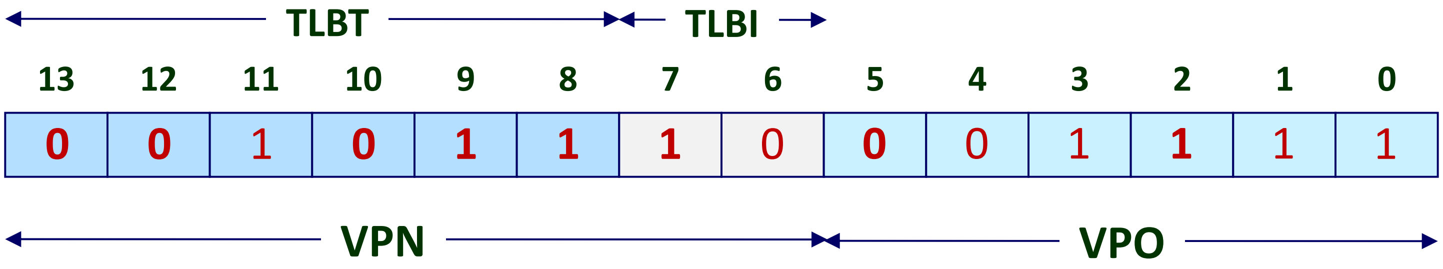
- Physical Address

| CT | | | | | | CI | | | | CO | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| PPN | | | | | | PPO | | | | | |

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

# Address Translation Example #2

- Virtual Address: 0x0B8F



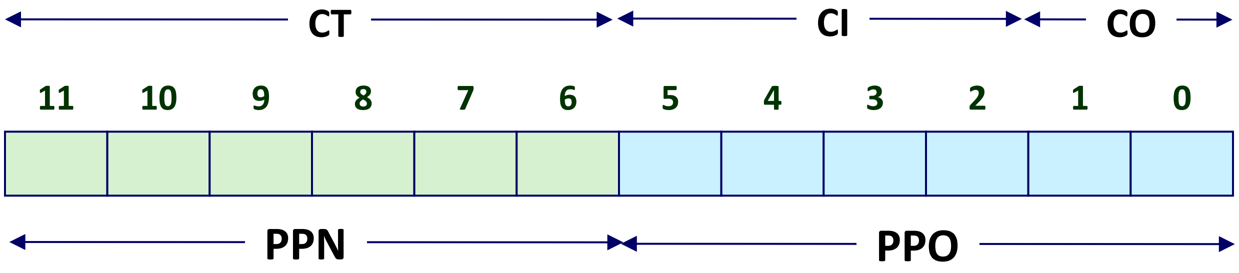
TLBT (bits 13–8) | TLBI (bits 7–6)

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

VPN (bits 13–6) | VPO (bits 5–0)

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

- Physical Address



CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

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

PPN (bits 11–6) | PPO (bits 5–0)

CO ___ CI___ CT ____ Hit? __ Byte: ____

# Address Translation Example #3

- Virtual Address: 0x0020

| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TLBT | | | | | | TLBI | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| VPN | | | | | | | | VPO | | | | | |

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

- Physical Address

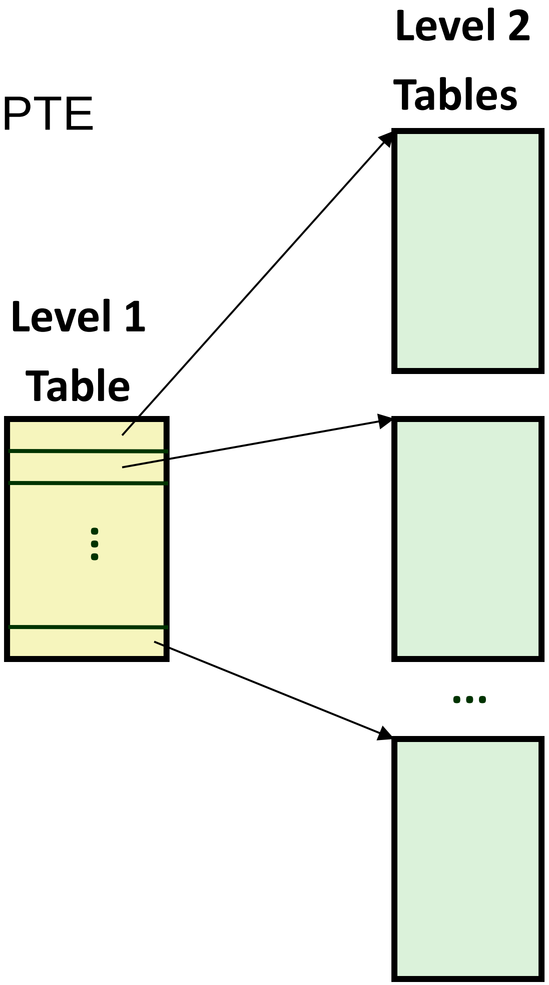
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CT | | | | | | CI | | | | CO | |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| PPN | | | | | | PPO | | | | | |

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

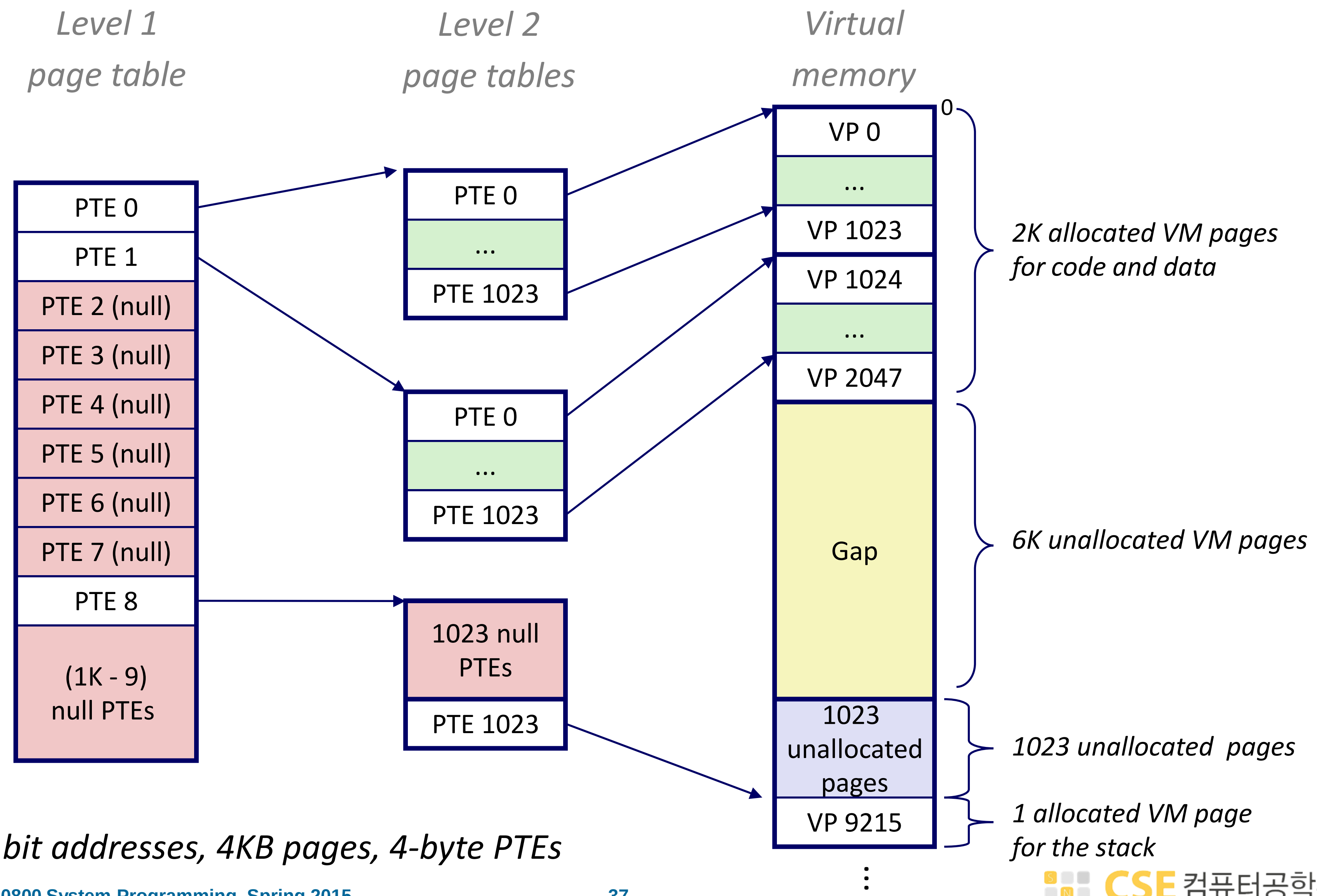
# Virtual Memory: Practice

# Multi-Level Page Tables

- Suppose:
  - 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 (always memory resident)
    - Level 2 table: each PTE points to a page (paged in and out like any other data)

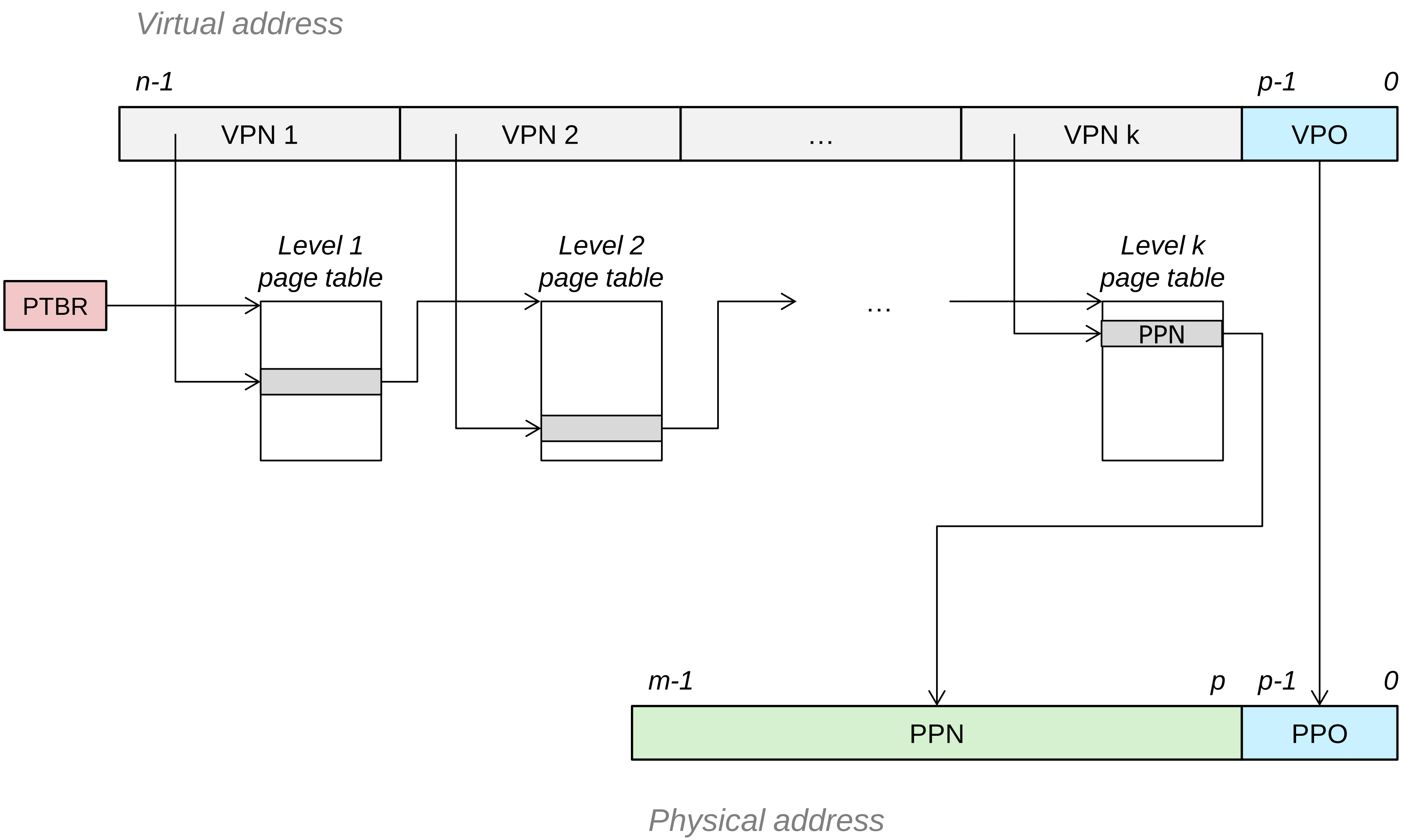


# A Two-Level Page Table Hierarchy

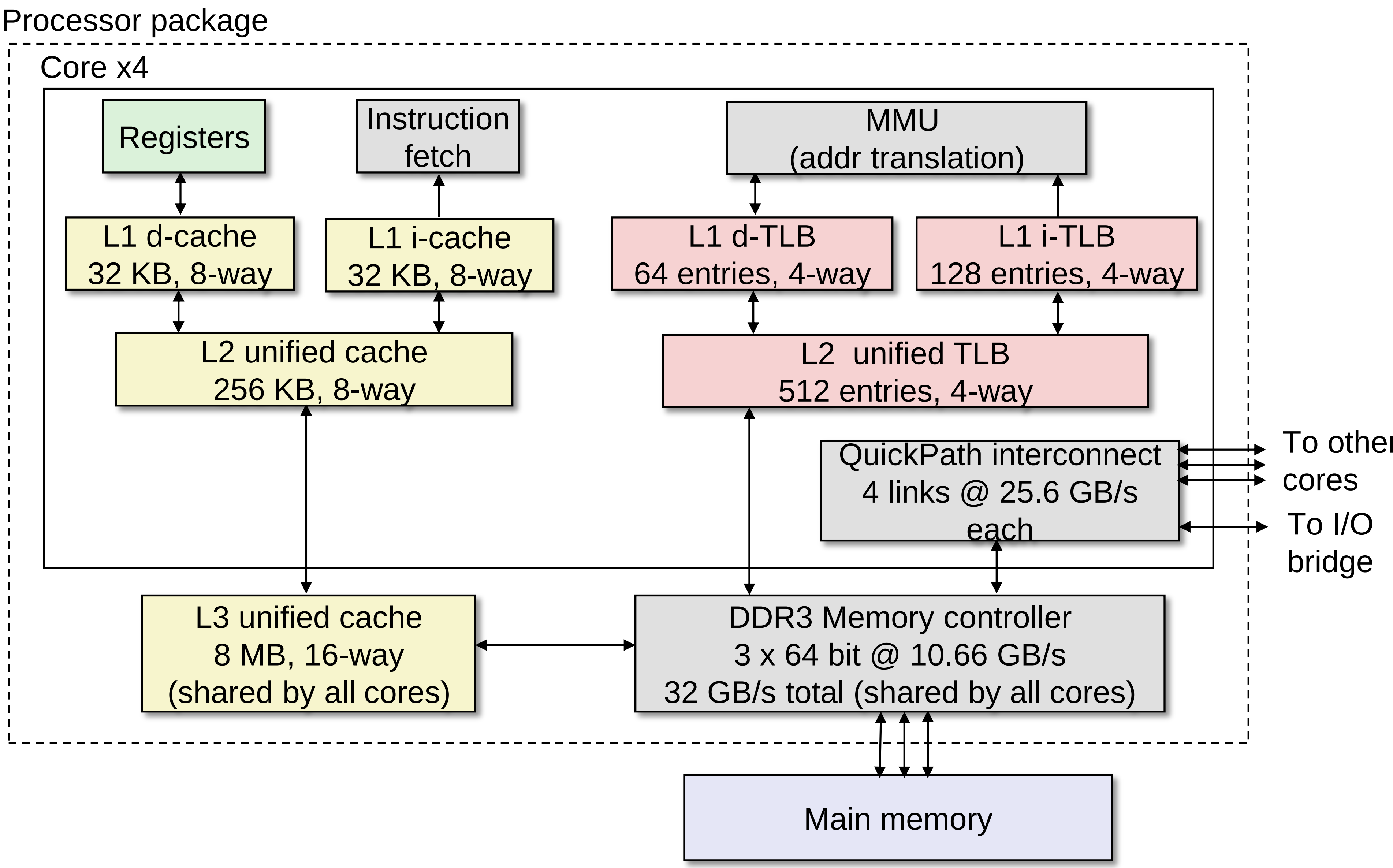


*32 bit addresses, 4KB pages, 4-byte PTEs*

# Address Translation with a Multi-level Page Table

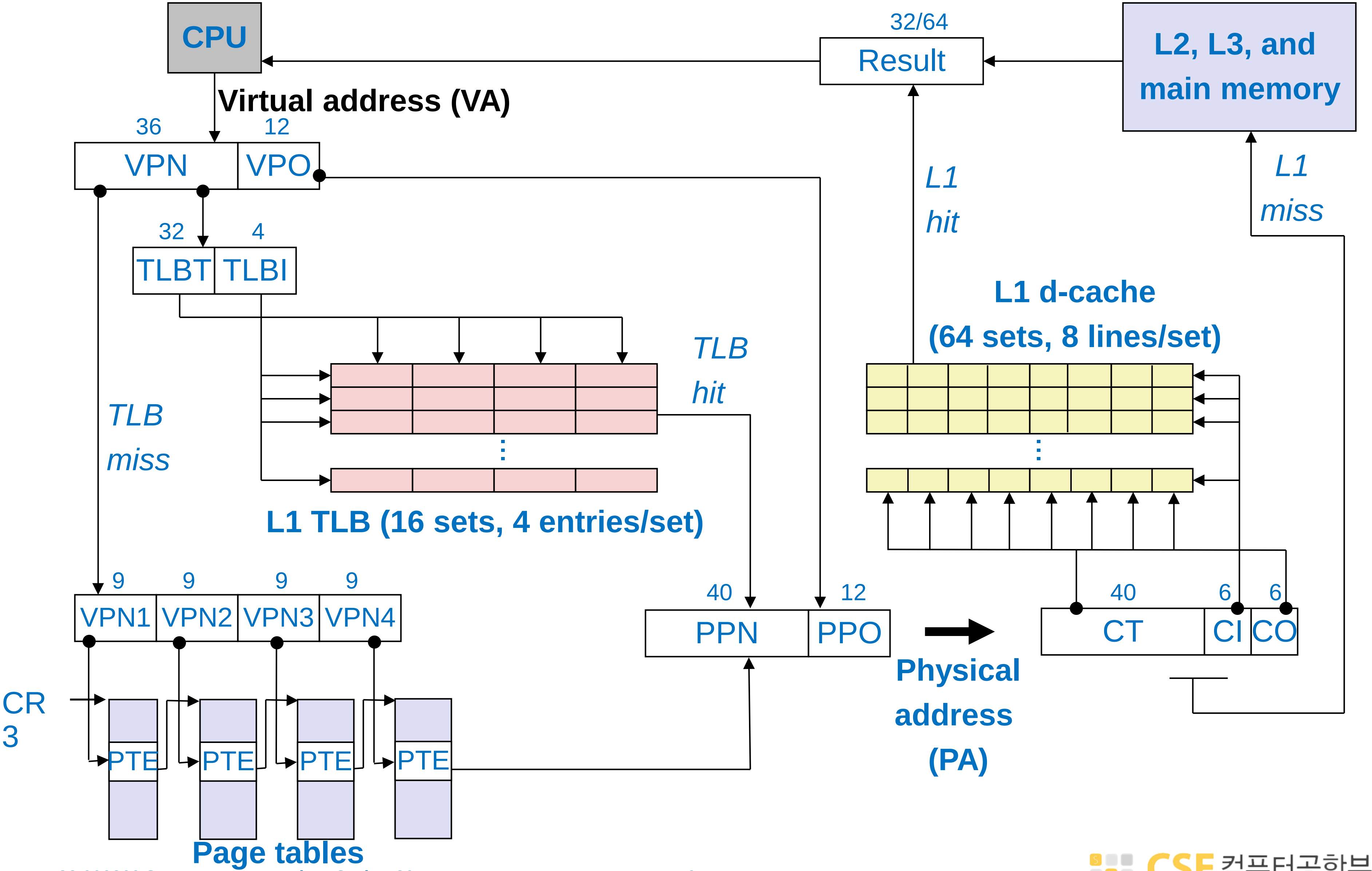


# Intel Core i7 Memory System



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

# End-to-end Core i7 Address Translation



# 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

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.

CD: Caching disabled or enabled 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).

G: Global page (don’t evict from TLB on task switch)

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

# 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

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

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

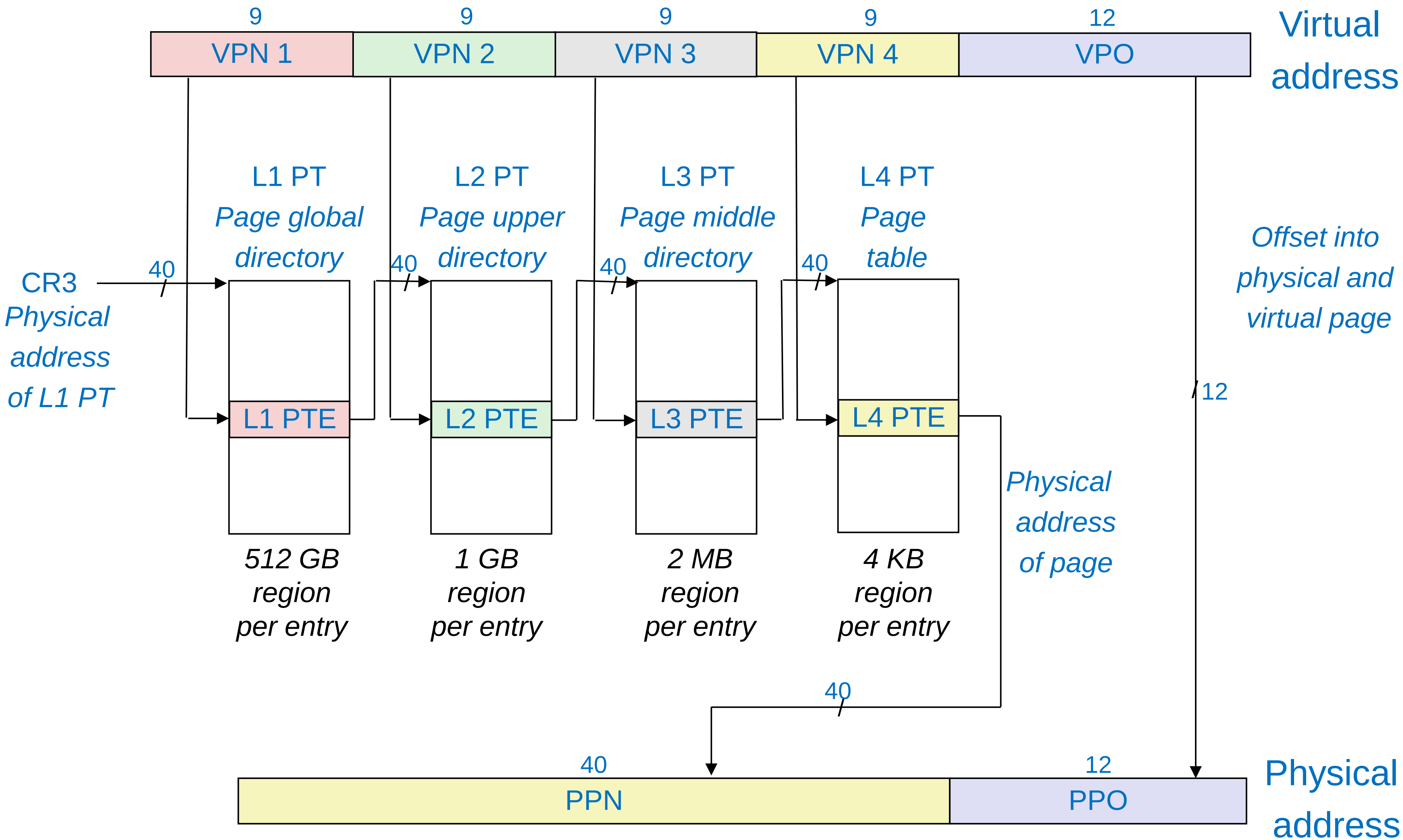
A: Reference bit (set by MMU on reads and writes, cleared by software)

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

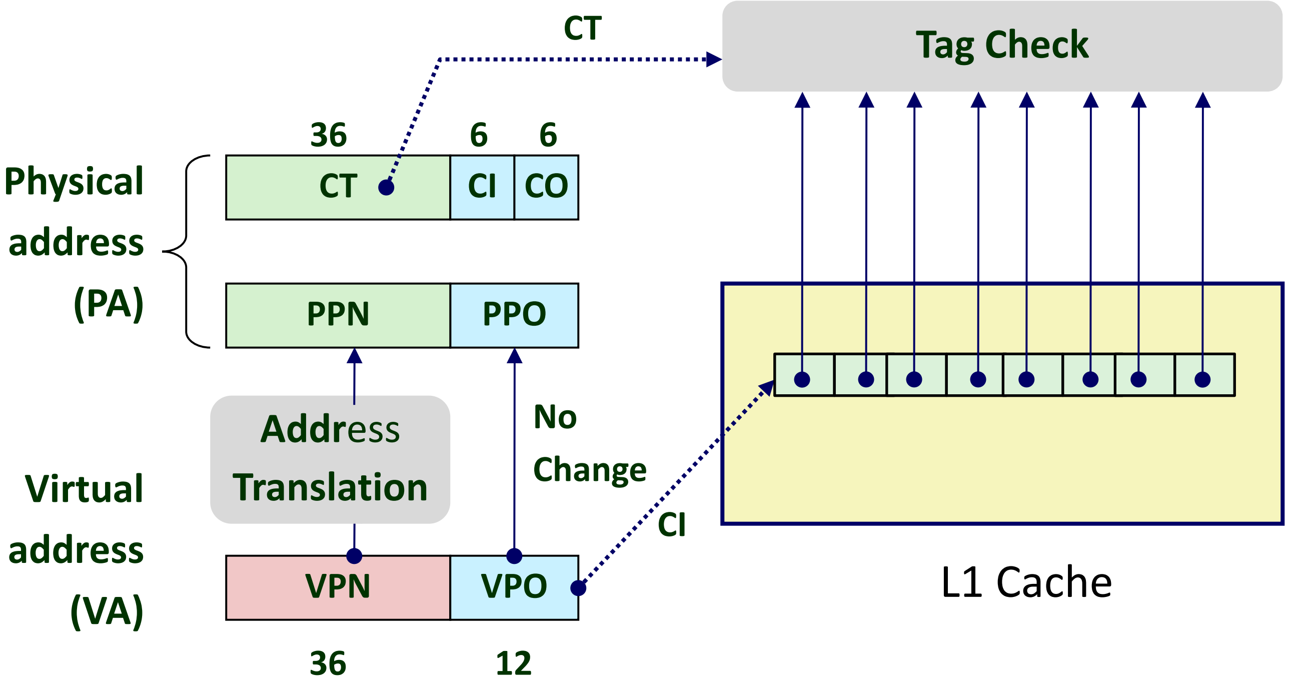
G: Global page (don't evict from TLB on task switch)

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

# Core i7 Page Table Translation

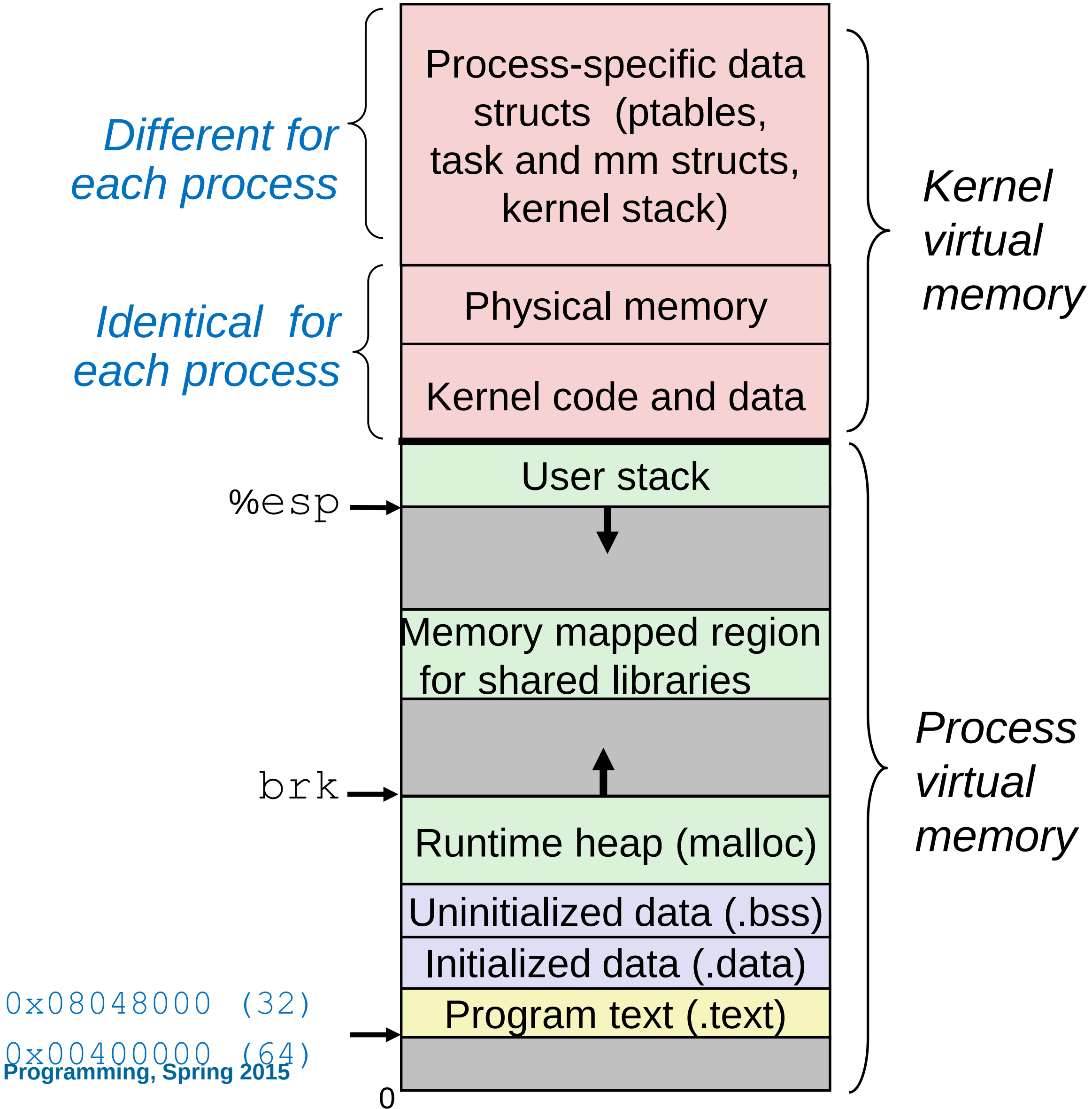


# Cute Trick for Speeding Up L1 Access



- 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

# Virtual Memory 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
%esp
User stack
Memory mapped region for shared libraries
brk
Runtime heap (malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
Process virtual memory
0x08048000 (32)
0x00400000 (64)
0

# 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.

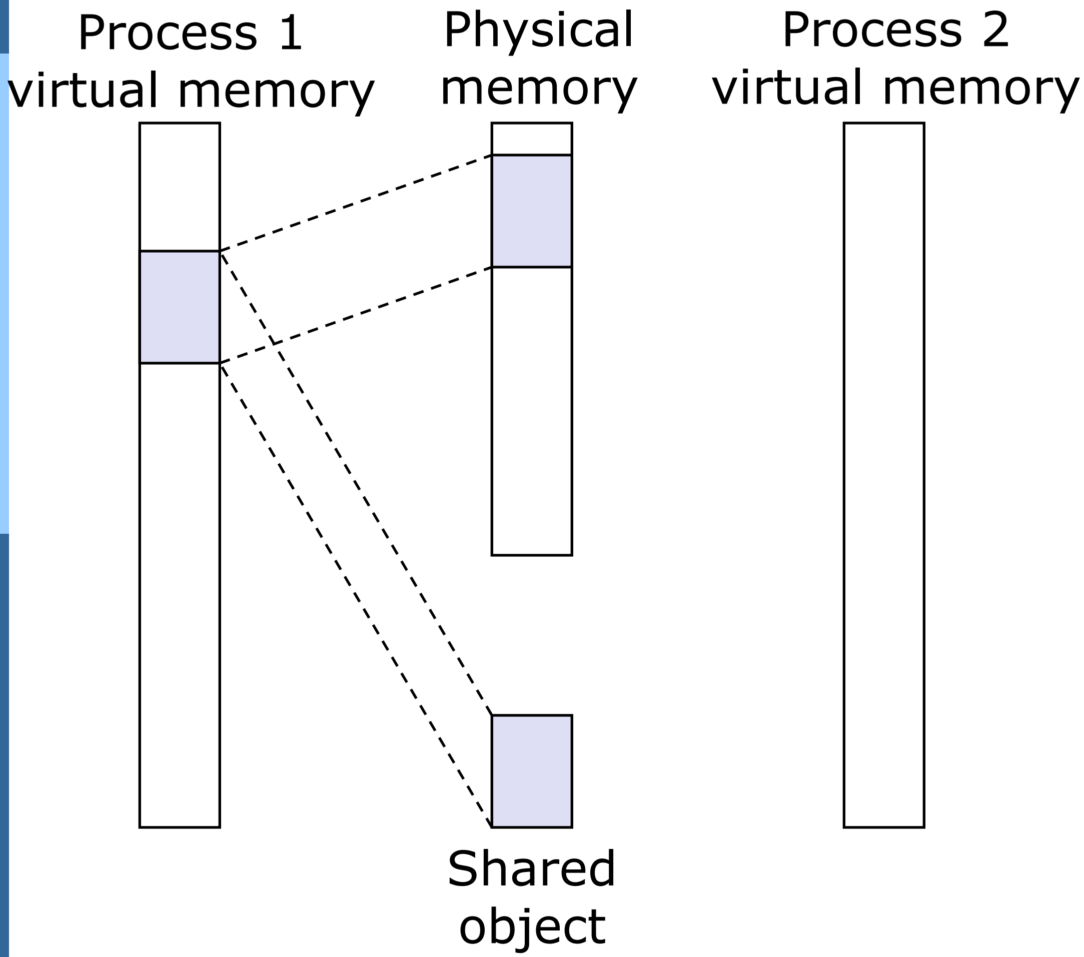
# Demand paging

- Key point: no virtual pages are copied into physical memory until they are referenced!
  - Known as demand paging

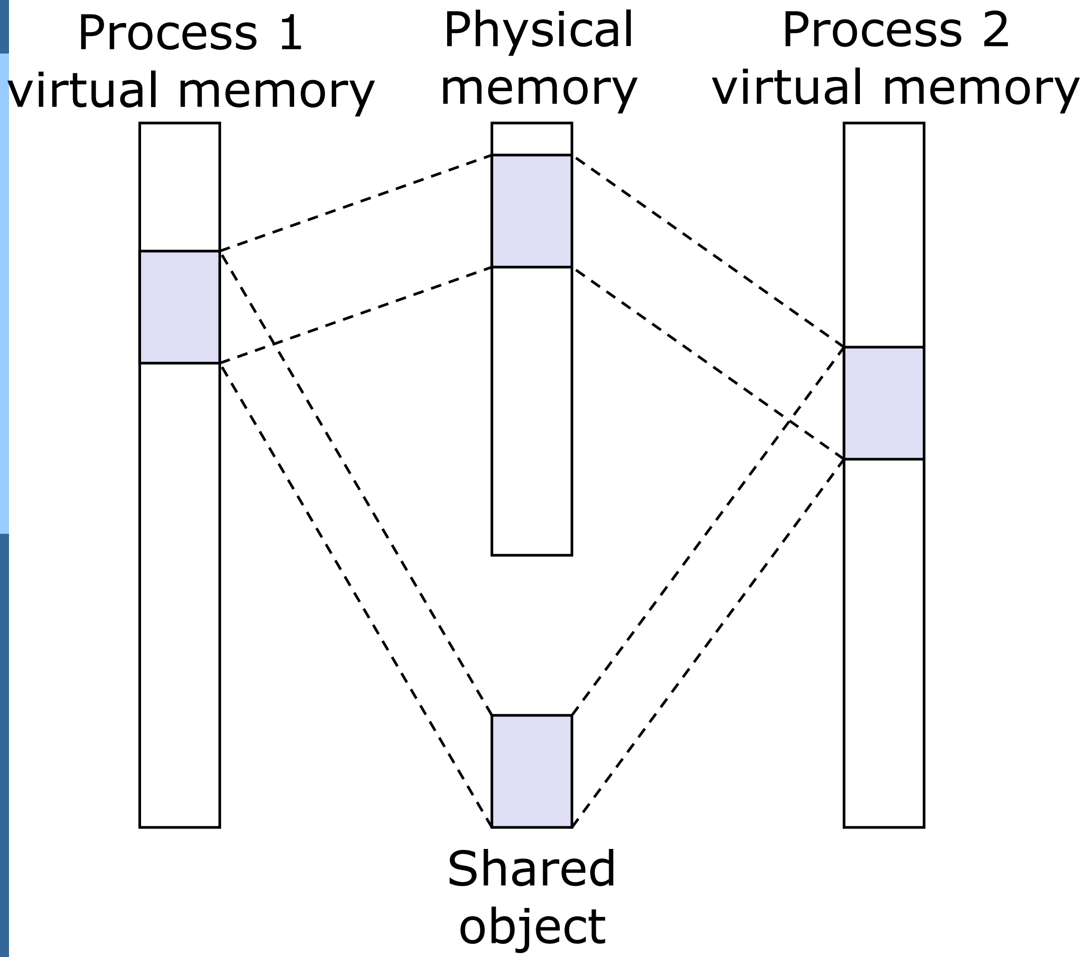
- Crucial for time and space efficiency

# Shared Memory Objects

- Process 1  maps the shared object.

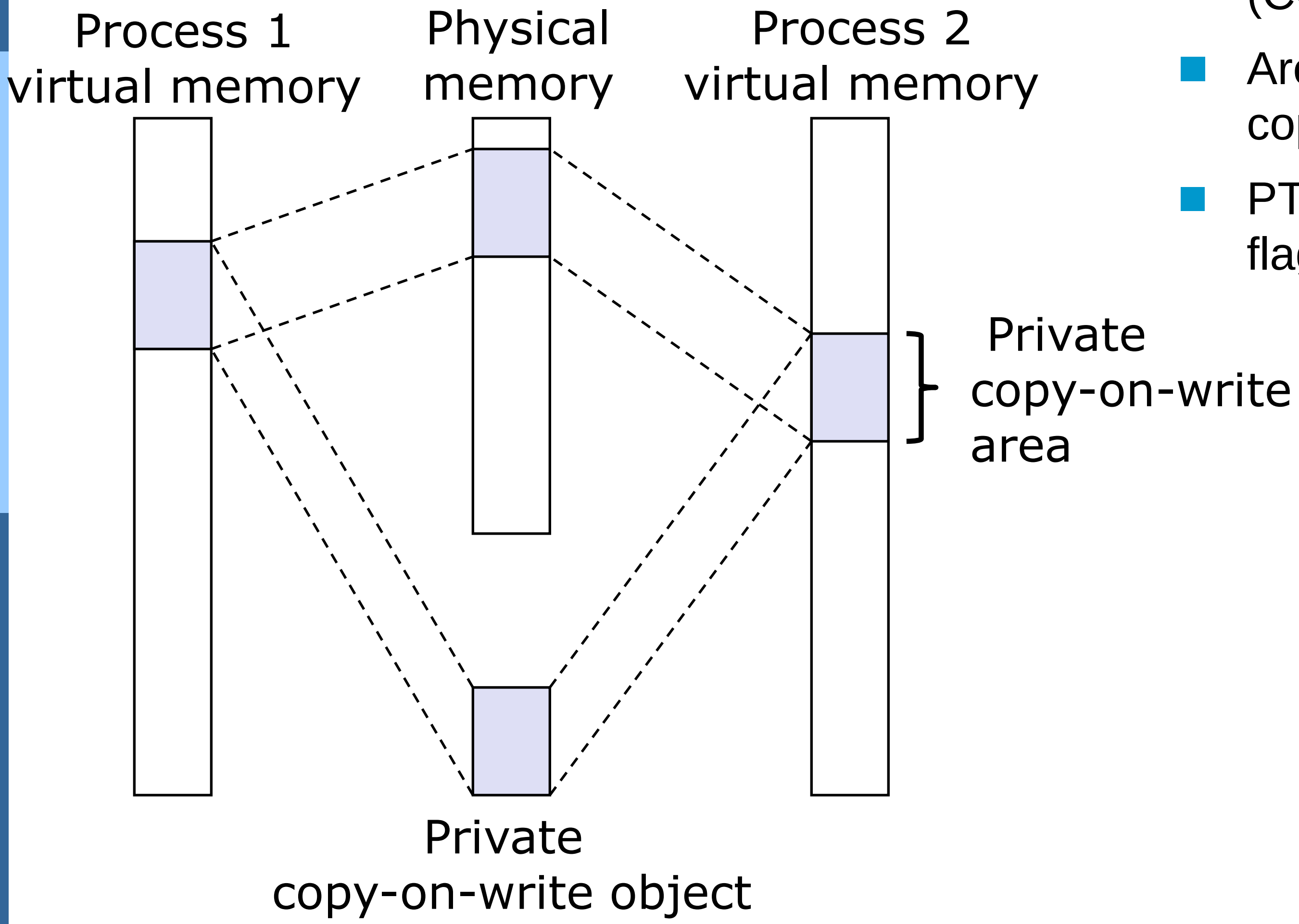


# Shared Memory Objects



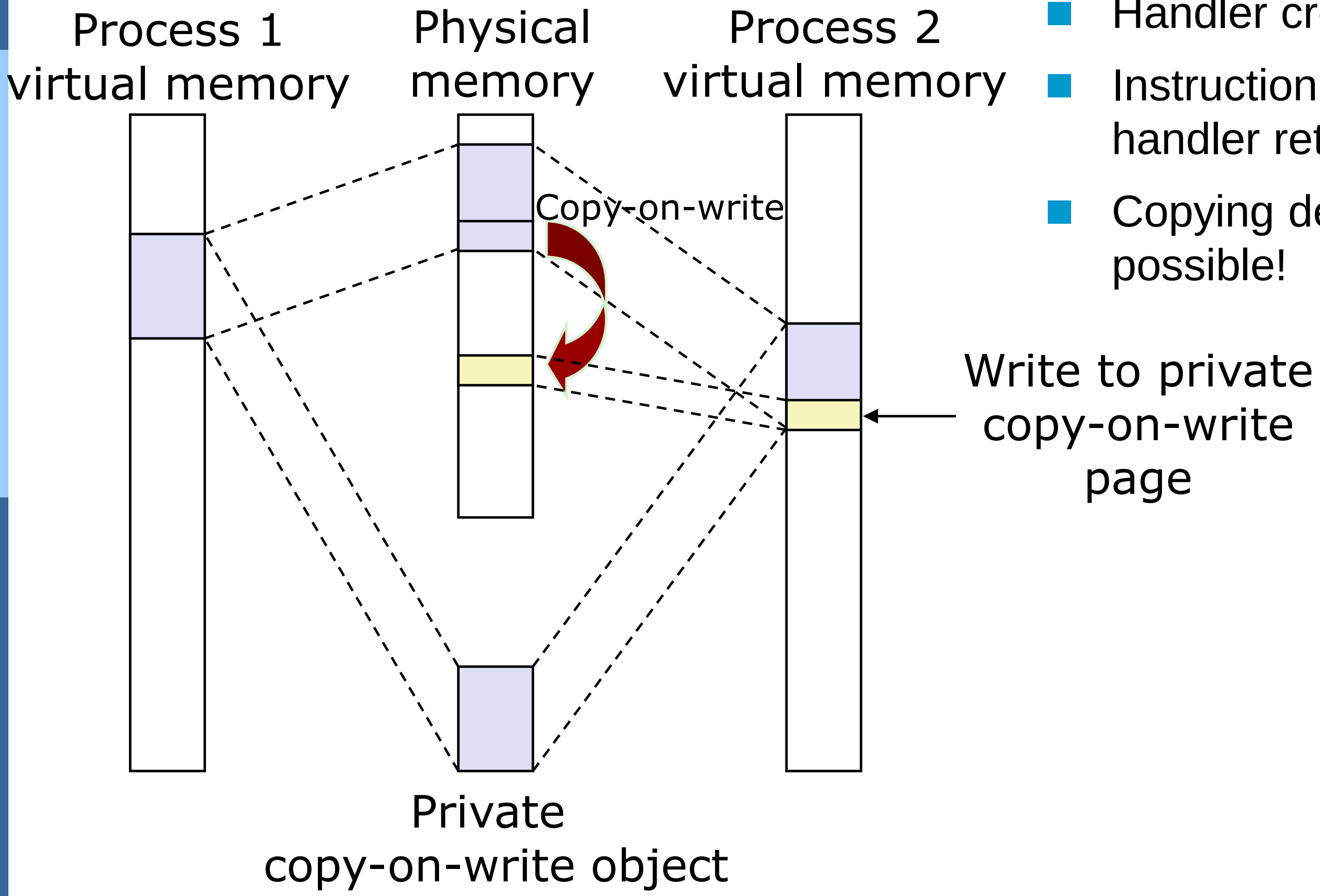
- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.

# 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

# 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!

# User-Level Memory Mapping

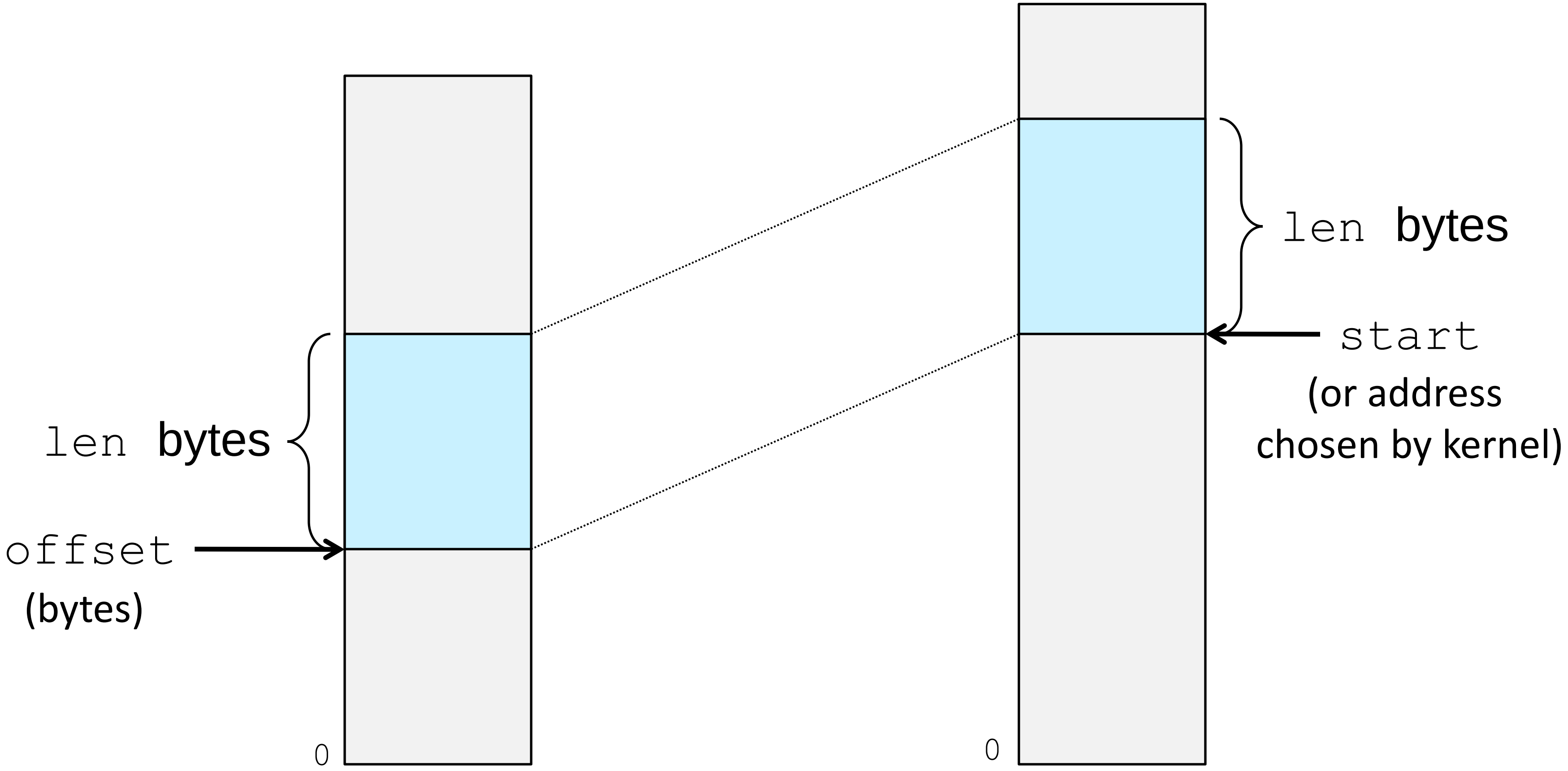
- void *mmap(void *start, int len, int prot, int flags, int fd, int offset)

- 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*

# Using mmap to Copy Files

- Copying without transferring data to user space!

```
#include "csapp.h“

/*
 * mmapcopy - uses mmap to copy
 *              file fd to stdout
 */
void mmapcopy(int fd, int size)
{
    /* ptr to mem-mapped VM area */
    char *bufp;

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

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

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

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