Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2021

Virtual Memory

Chap. 5.7 – 5.8, 5.13, 5.16 – 5.17

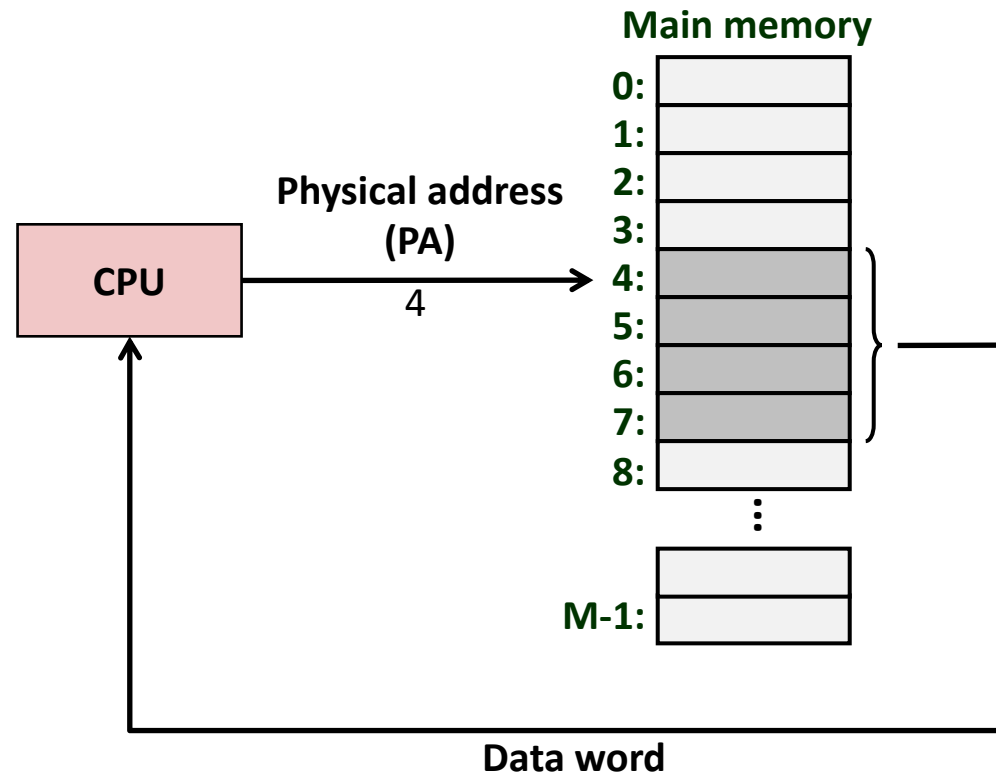# Virtualizing Memory

- Example

```
#include <stdio.h>

int n = 0;

int main ()
{
    n++;
    printf ("&n = %p, n = %d\n", &n, n);
}

% ./a.out
&n = 0x0804a024, n = 1
% ./a.out
&n = 0x0804a024, n = 1
```
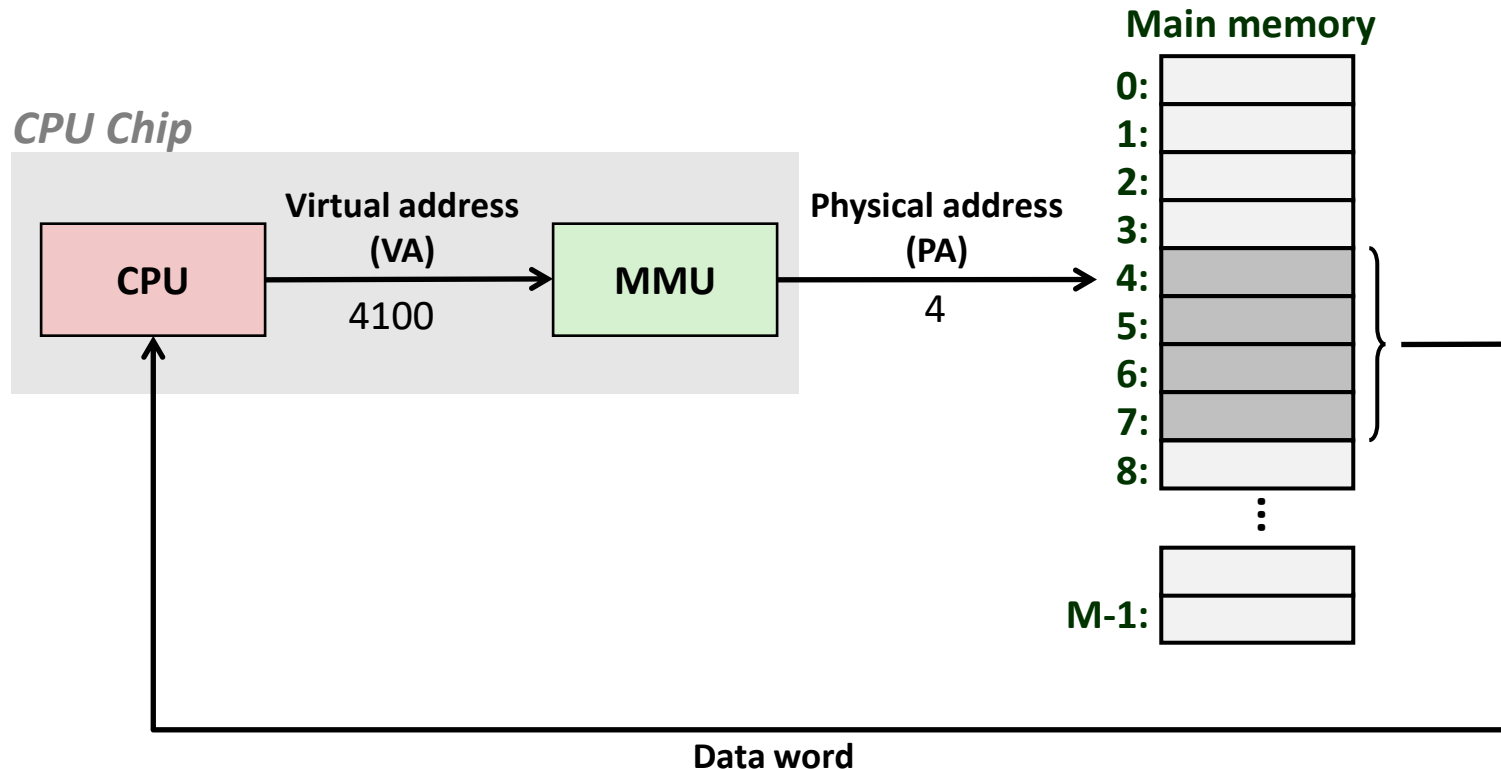
Physical Addressing

- Used in "simple" systems like embedded microcontrollers
  - Cars, elevators, digital cameras, etc.

Main memory

```
0:
1:
2:
3:
4:
5:
6:
7:
8:
   ⋮
M-1:
```

CPU

Physical address (PA)

4

Data word

Virtual Addressing

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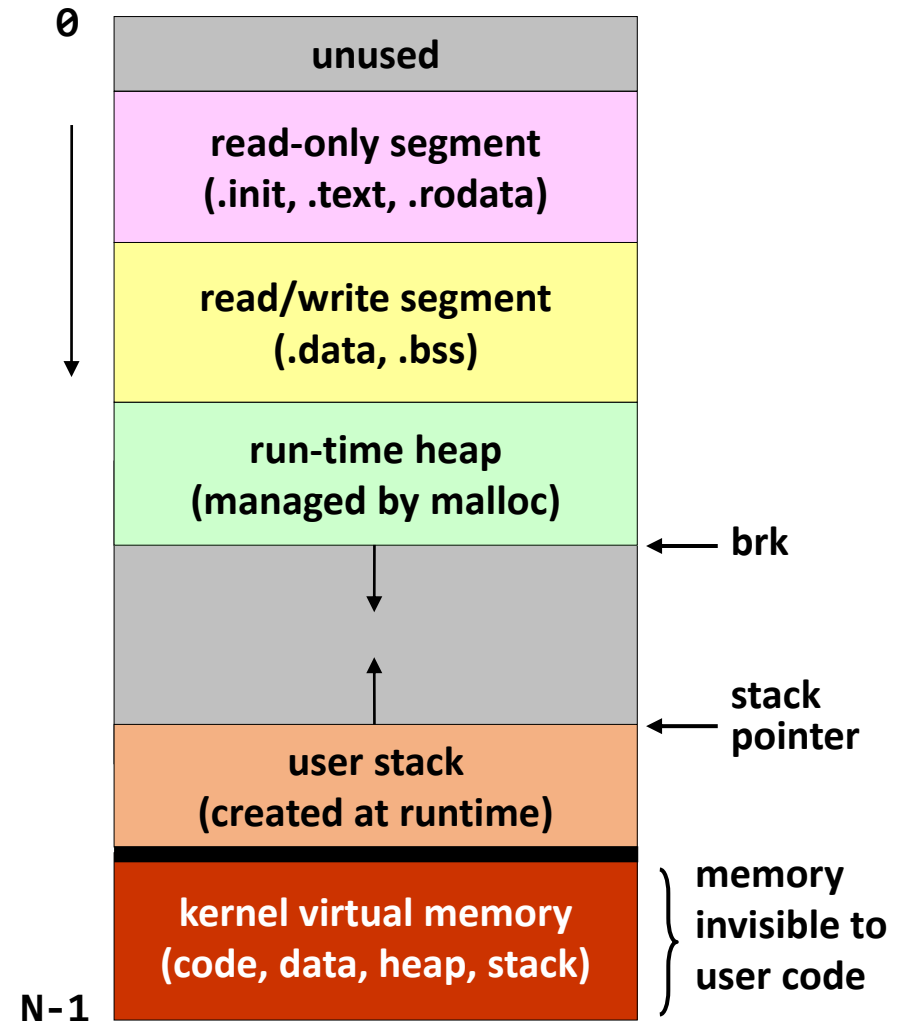- One of the great ideas in computer science

Virtual Memory

- **Use main memory as a "cache" for secondary (disk) storage**
  - Managed jointly by CPU hardware and the operating system

- **Programs share main memory**
  - Each gets a private virtual address space holding its frequently used code and data
  - Use virtual addresses for memory references
  - Virtual address space is protected from other processes
  - Lazy allocation: physical memory is dynamically allocated or released on demand

- **CPU and OS translate virtual addresses to physical addresses**
  - VM "block" is called a page
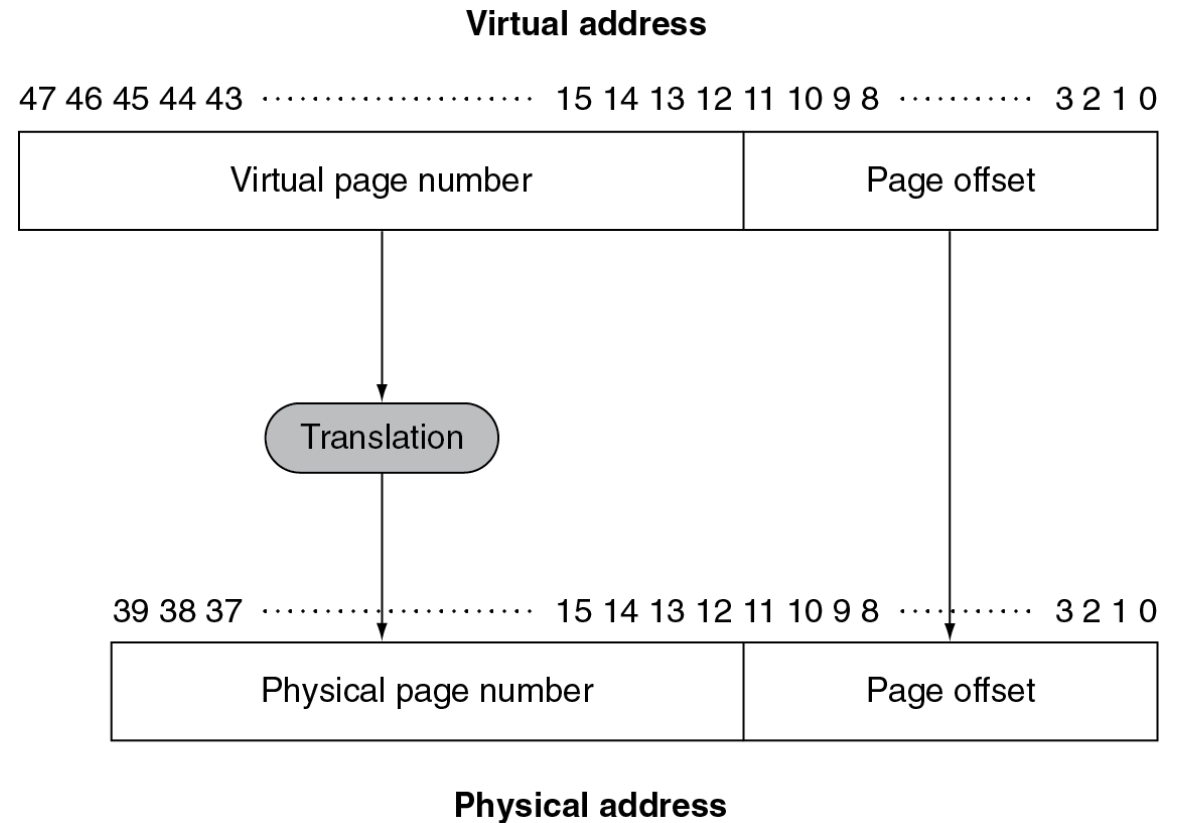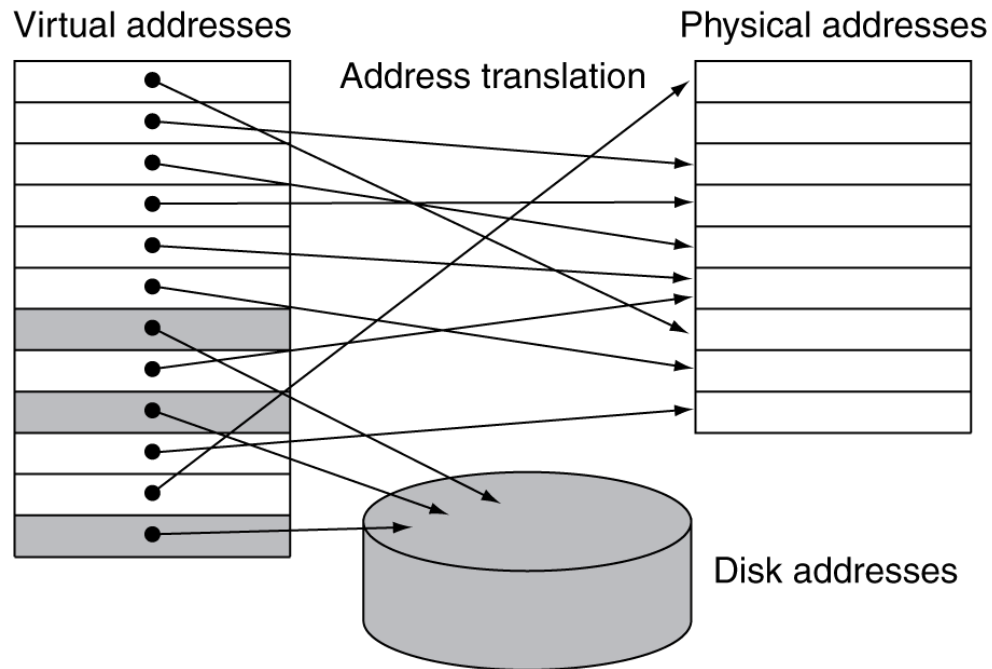 - VM translation "miss" is called a page fault

(Virtual) Address space

- **Process' abstract view of memory**
  - OS provides illusion of private address space to each process
  - Contains all of the memory state of the process

- **Static area: allocated on exec()**
  - Code & Data

- **Dynamic area: allocated at runtime**
  - Can grow or shrink
  - Heap & Stack

0

unused

read-only segment
(.init, .text, .rodata)

read/write segment
(.data, .bss)

run-time heap
(managed by malloc)

← brk

stack pointer →

user stack
(created at runtime)

kernel virtual memory
(code, data, heap, stack)

} memory invisible to user code

N-1

Address Translation

- Fixed-size pages (e.g., 4KB)

Virtual address

47 46 45 44 43 ················· 15 14 13 12 11 10 9 8 ·········· 3 2 1 0

Virtual page number	Page offset

Translation

39 38 37 ················· 15 14 13 12 11 10 9 8 ········· 3 2 1 0

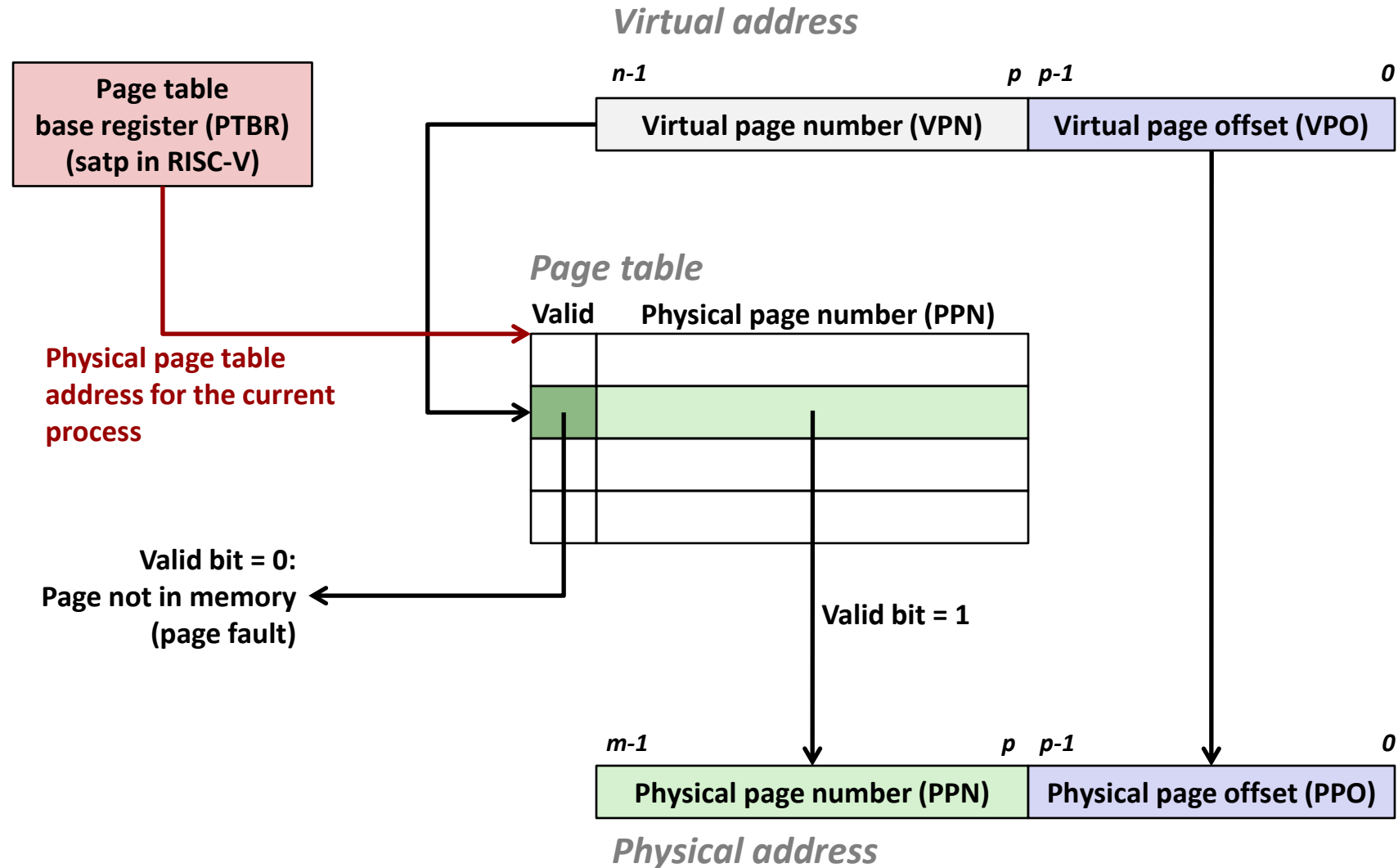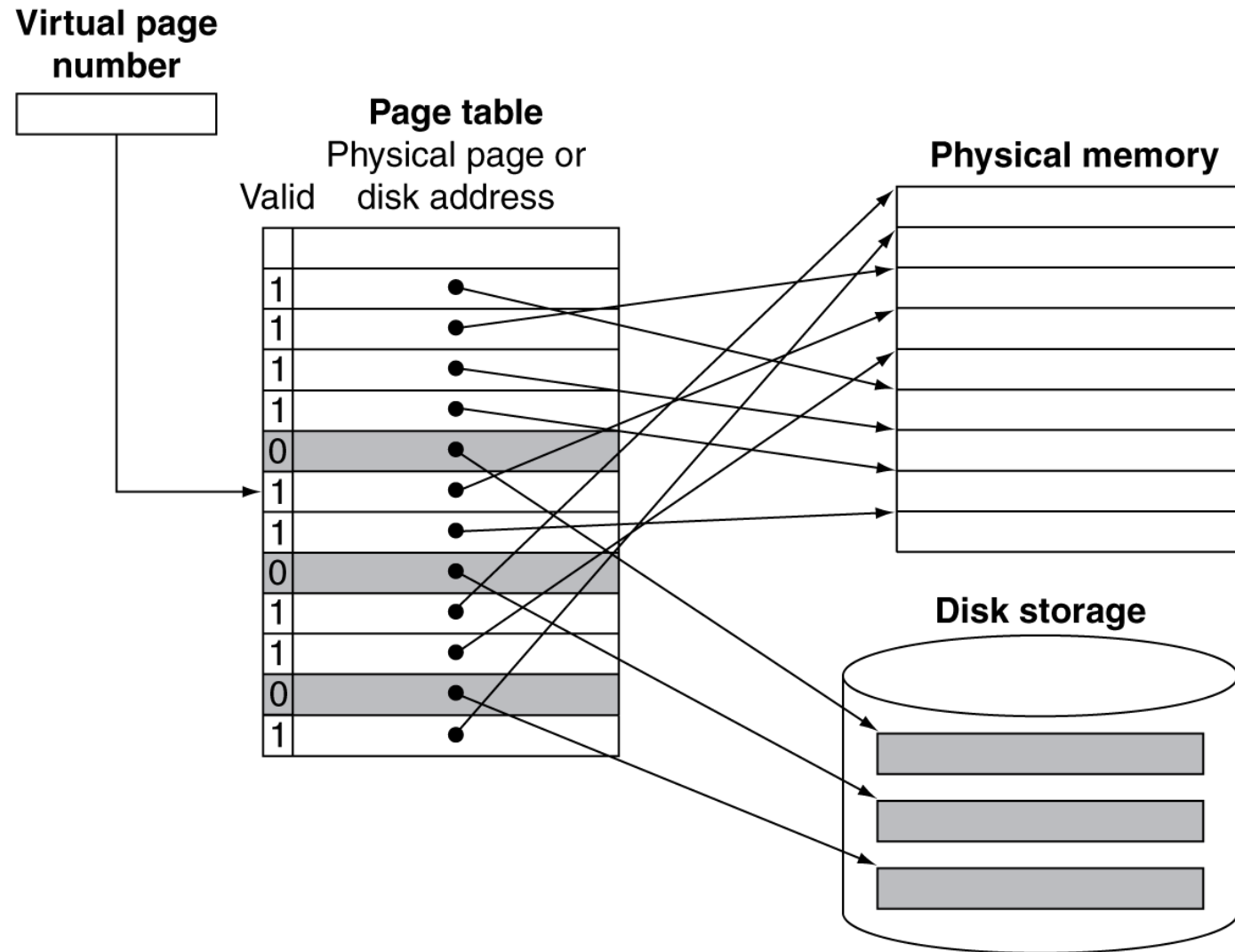| Physical page number	Page offset

Physical address

Page Tables

- **Stores placement information**
  - Array of page table entries (PTEs), indexed by virtual page number
  - Page table register in CPU points to page table in physical memory

- **If page is present in memory**
  - PTE stores the physical page number
  - Plus other status bits (protection, referenced, dirty, …)

- **If page is not present**
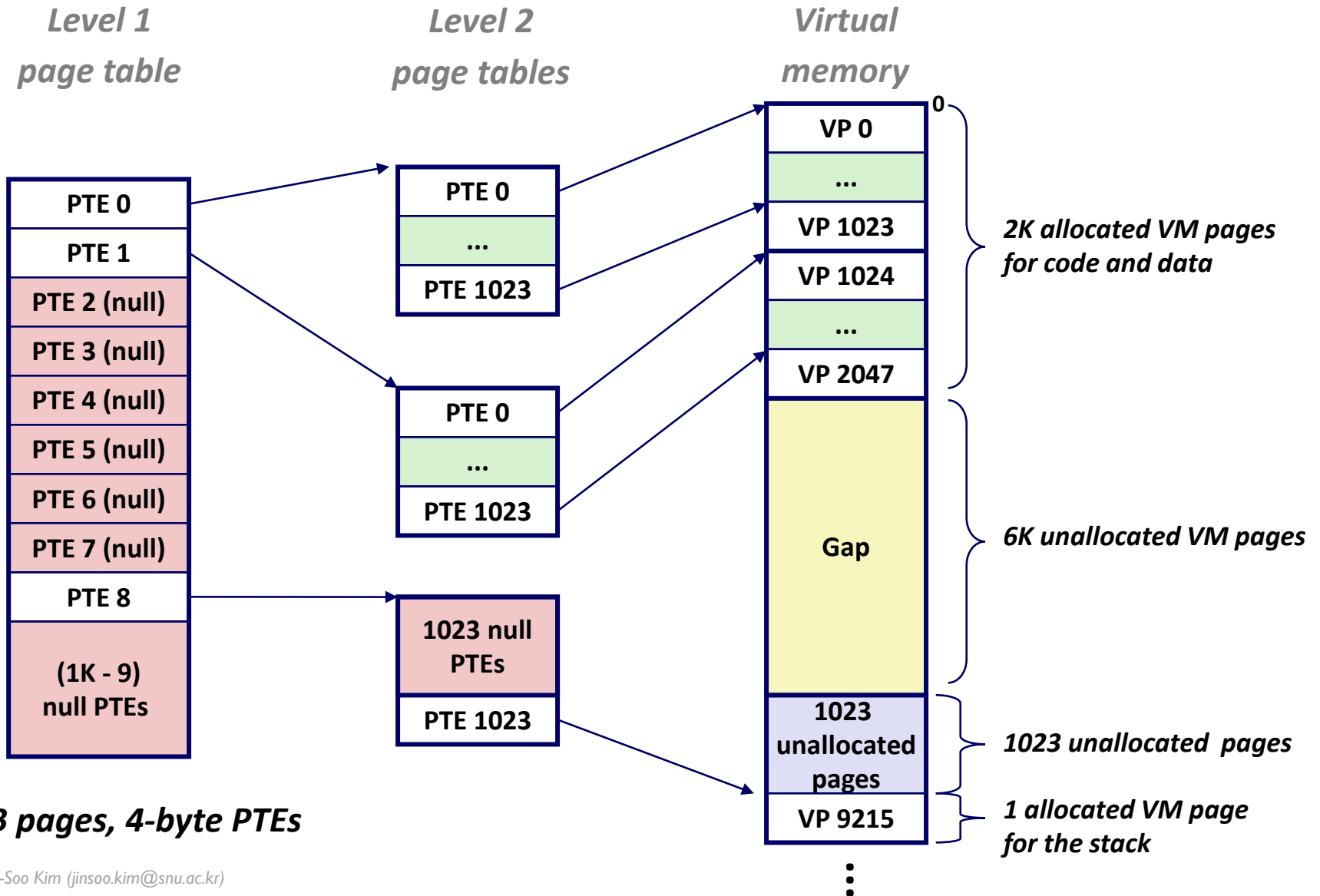 - PTE can refer to location in swap space on disk

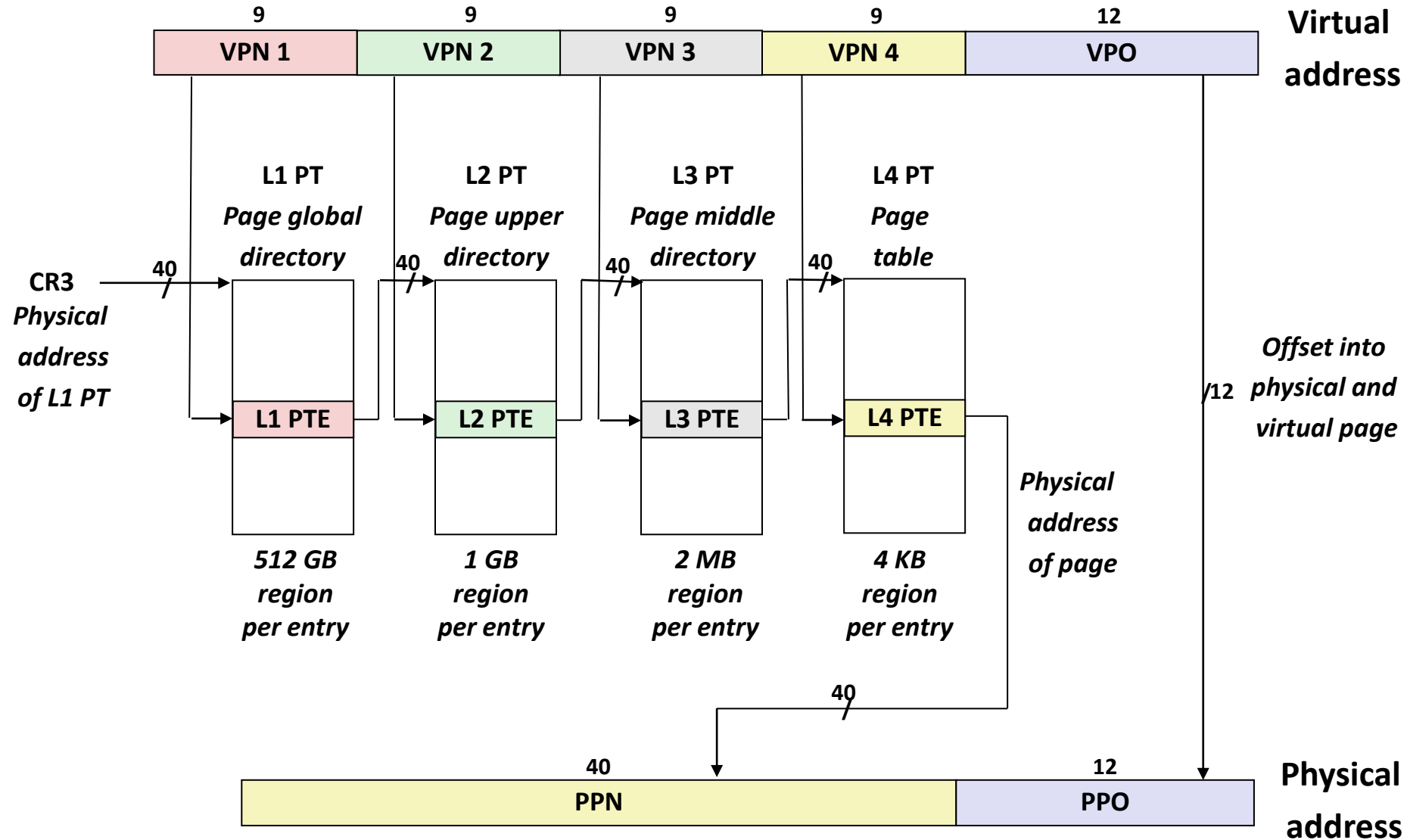# Address Translation with a Page Table

Mapping Pages to Storage

A Two-Level Page Table Hierarchy

Level 1
page table

Level 2
page tables

Virtual
memory

PTE 0	
PTE 1	
PTE 2 (null)	
PTE 3 (null)	
PTE 4 (null)	
PTE 5 (null)	
PTE 6 (null)	
PTE 7 (null)	
PTE 8	
(1K - 9) null PTEs	

PTE 0
...
PTE 1023

PTE 0
...
PTE 1023

1023 null PTEs
PTE 1023

VP 0
...
VP 1023
VP 1024
...
VP 2047

Gap

1023 unallocated pages

VP 9215

0

2K allocated VM pages for code and data

6K unallocated VM pages

1023 unallocated pages

1 allocated VM page for the stack

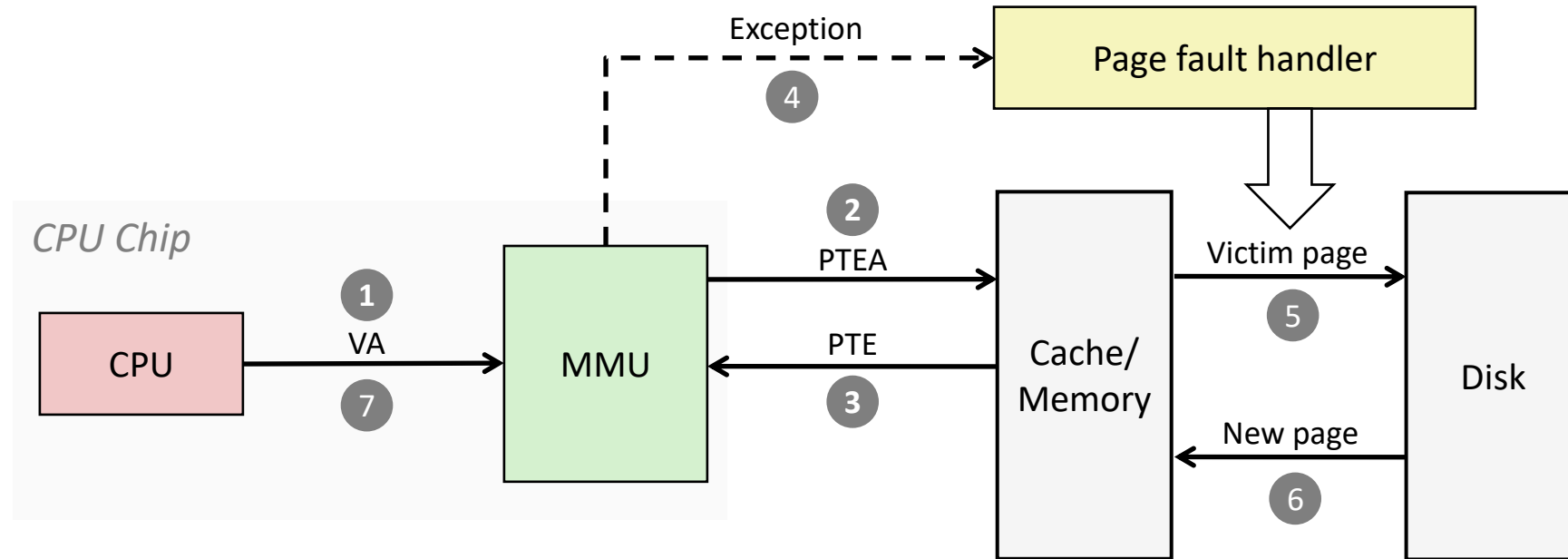**32 bit addresses, 4KB pages, 4-byte PTEs**

Core i7 Page Table Translation

Address Translation: Page Hit
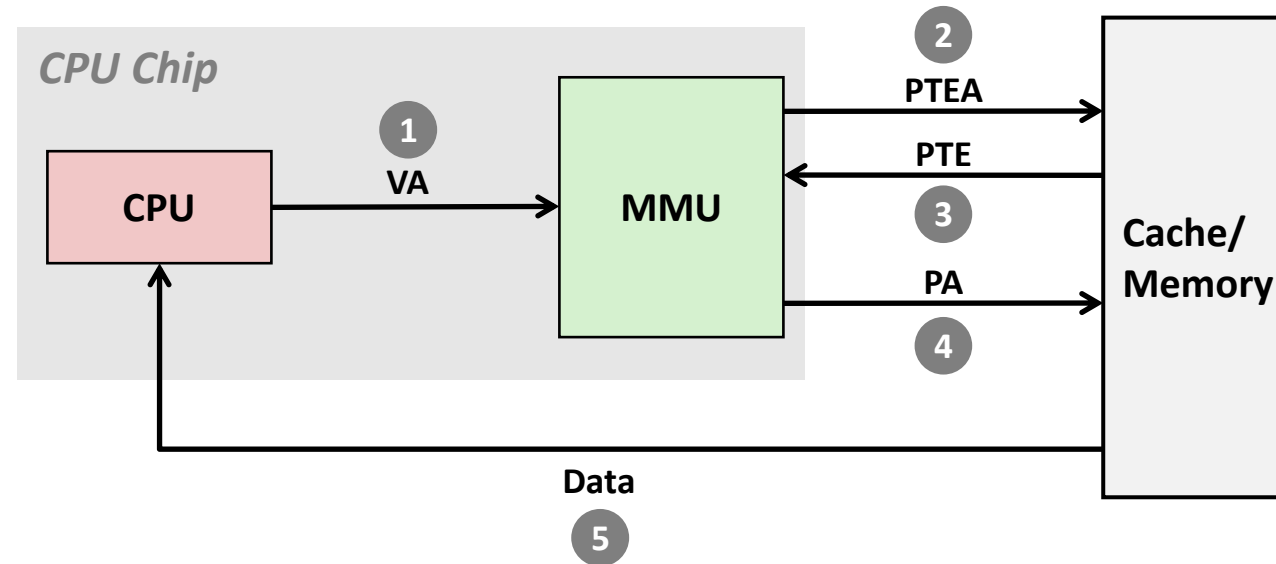
1) Processor sends virtual address to MMU

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

TLB

- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access

- But access to page tables has good locality
  - Use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16 – 512 PTEs, 0.5 – 1 cycle for hit, 10 – 100 cycles for miss, 0.01% – 1% miss rate
  - Misses could be handled by hardware or software

Fast Translation Using a TLB
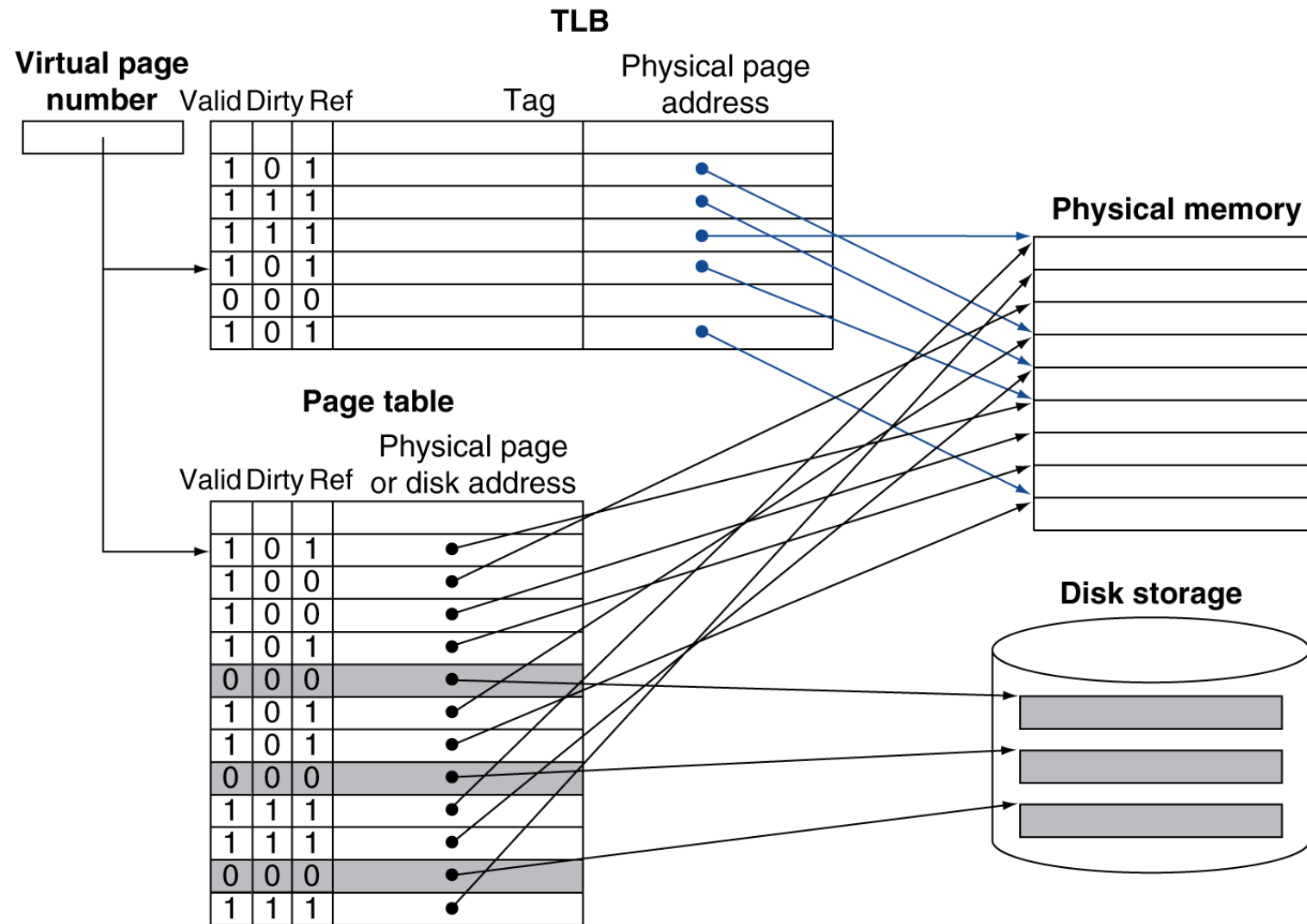
TLB Misses

- **If page is in memory**

  - Load the PTE from memory and retry

  - Could be handled in hardware: can get complex for more complicated page table structures

  - Or in software: raise a special exception with optimized handler ("TLB miss handler")

  - Modern CPUs usually implement "page table walk" in hardware

- **If page is not in memory (page fault)**

  - OS handles fetching the page and updating the page table

  - Then restart the faulting instruction
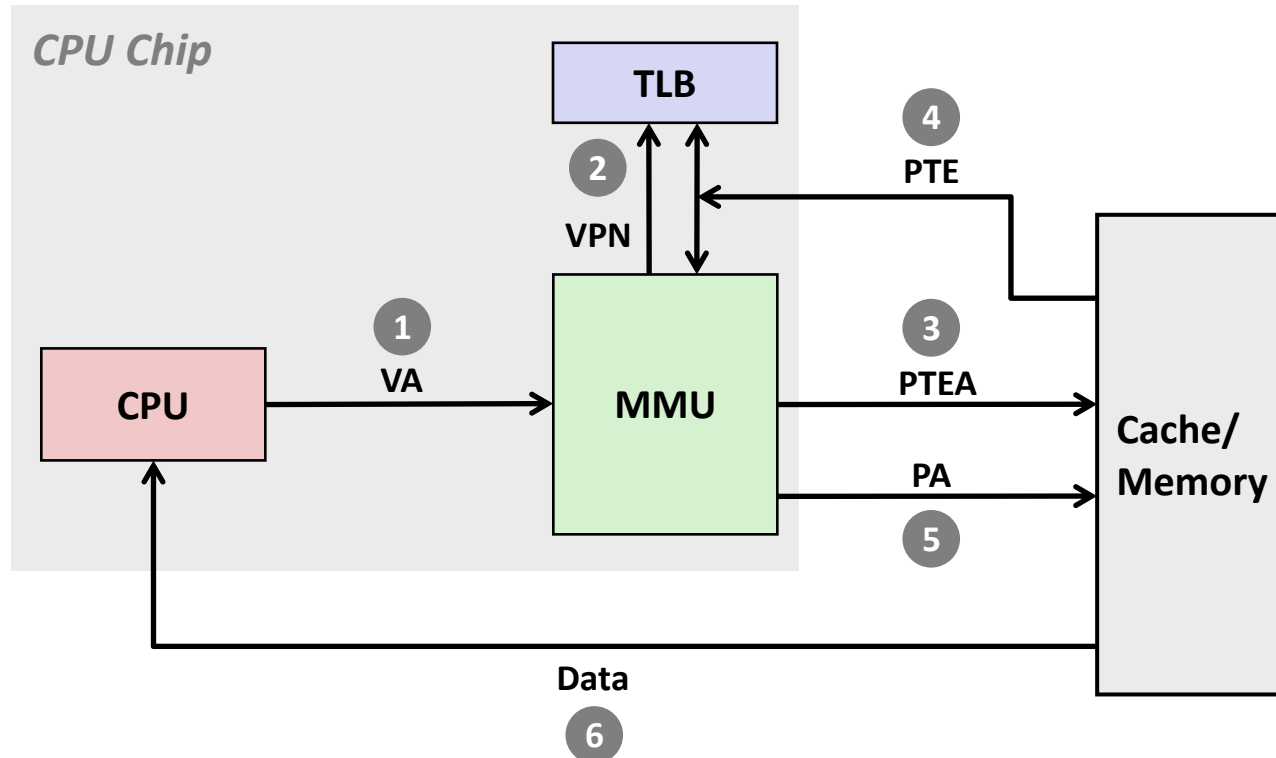
TLB Hit

- A TLB hit eliminates a memory access
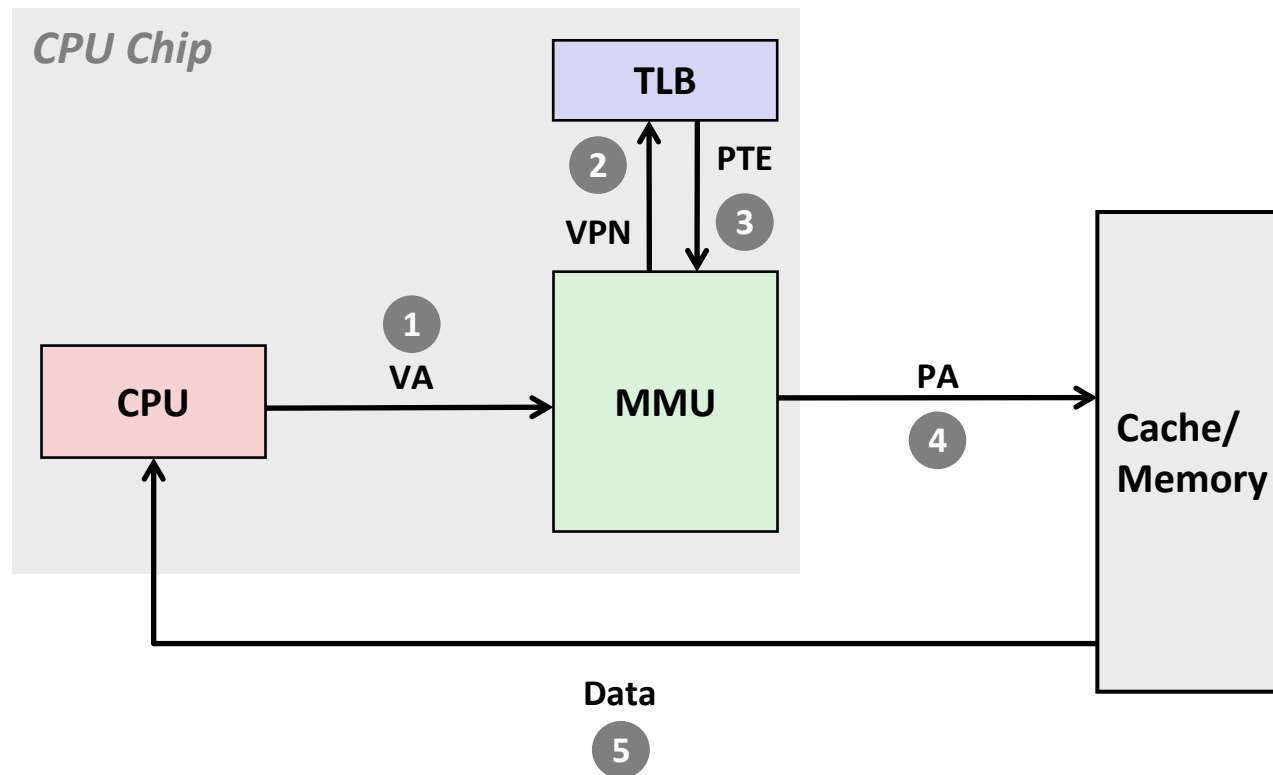
TLB Miss

- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare. Why?

Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code

- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

Replacement and Writes

- **To reduce page fault rate, prefer least-recently used (LRU) replacement**
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently

- **Disk writes take millions of cycles**
  - Block at once, not individual locations
  - Use write-back: write-through is impractical
  - Dirty it in PTE set when page is written

Page Fault Handler

- Use faulting virtual address to find PTE

- Locate page on disk

- Choose page to replace
  - If dirty, write to disk first

- Read page into memory and update page table

- Make process runnable again
  - Restart from faulting instruction

Memory Protection

- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., `ecall` in RISC-V)

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance

From CPU to Memory

- **Physically addressed cache**
  - Allows multiple processes to have blocks in cache
  - Allows multiple processes to share pages
  - Address translation is on the critical path
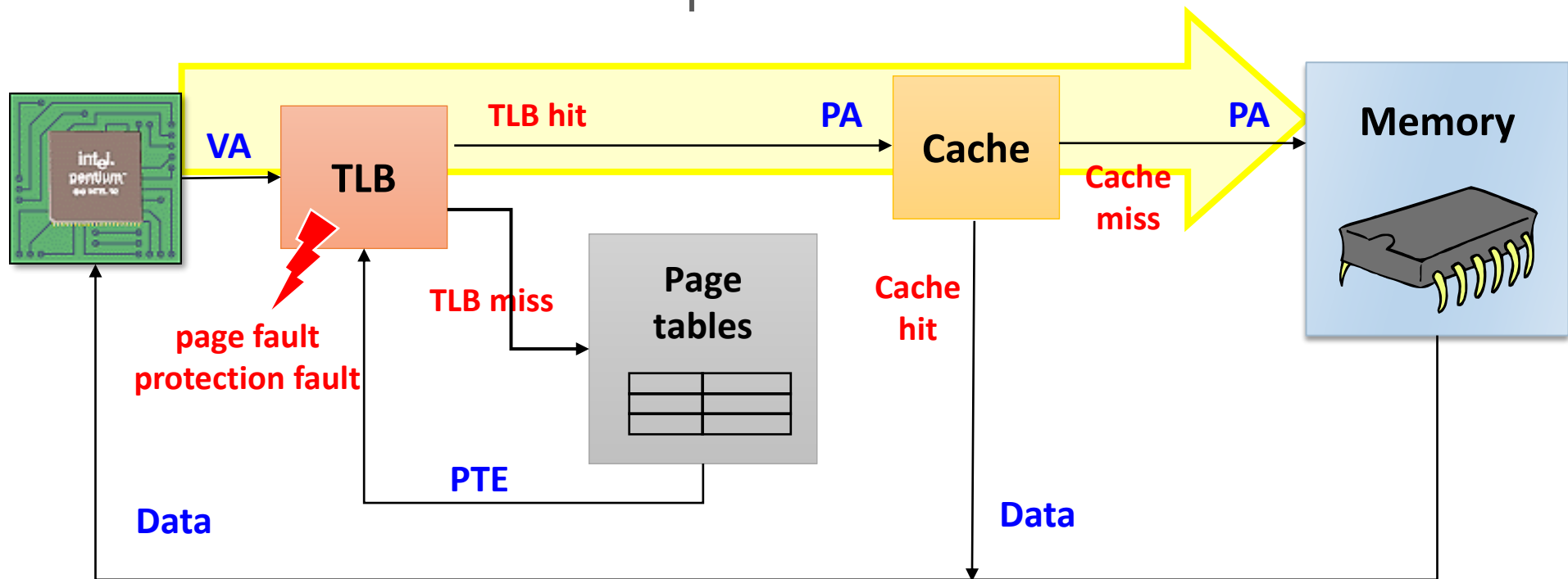
Intel Core i7 Memory System
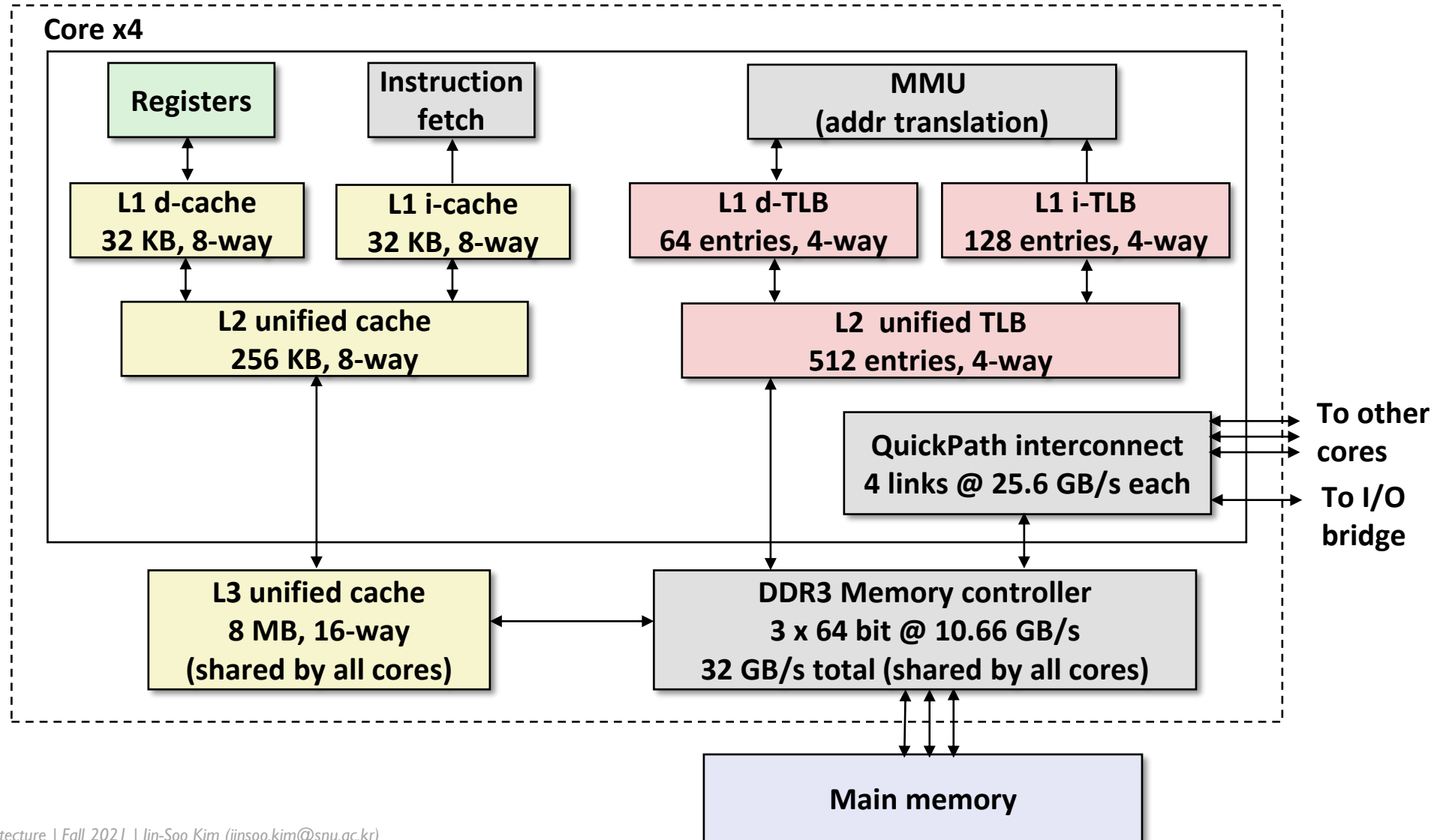
Memory Hierarchy Principles

Memory Hierarchy Principles

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching

- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- Determined by associativity
  - Direct mapped (1-way associative) – one choice for placement
  - *n*-way set associative – *n* choices within a set
  - Fully associative – any location

- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	*n*
Fully associative	Search all entries	#entries
	Full lookup table	0

- **Hardware caches**
  - Reduce comparisons to reduce cost

- **Virtual memory**
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

■ **Choice of entry to replace on a miss**

- Least recently used (LRU)
  - Complex and costly hardware for high associativity

- Random
  - Close to LRU, easier to implement

■ **Virtual memory**

- LRU approximation with hardware support

Write Policy

- **Write-through**
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer

- **Write-back**
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state

- **Virtual memory**
  - Only write-back is feasible, given disk write latency

Sources of Misses (Three Cs)

- **Compulsory** misses (or cold-start misses)
  - First access to a block

- **Capacity** misses
  - Due to finite cache size
  - A replaced block is later accessed again

- **Conflict** misses (or collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Sources of Misses: Example

- **SPEC2000 benchmarks**
  - Compulsory misses: 0.006%
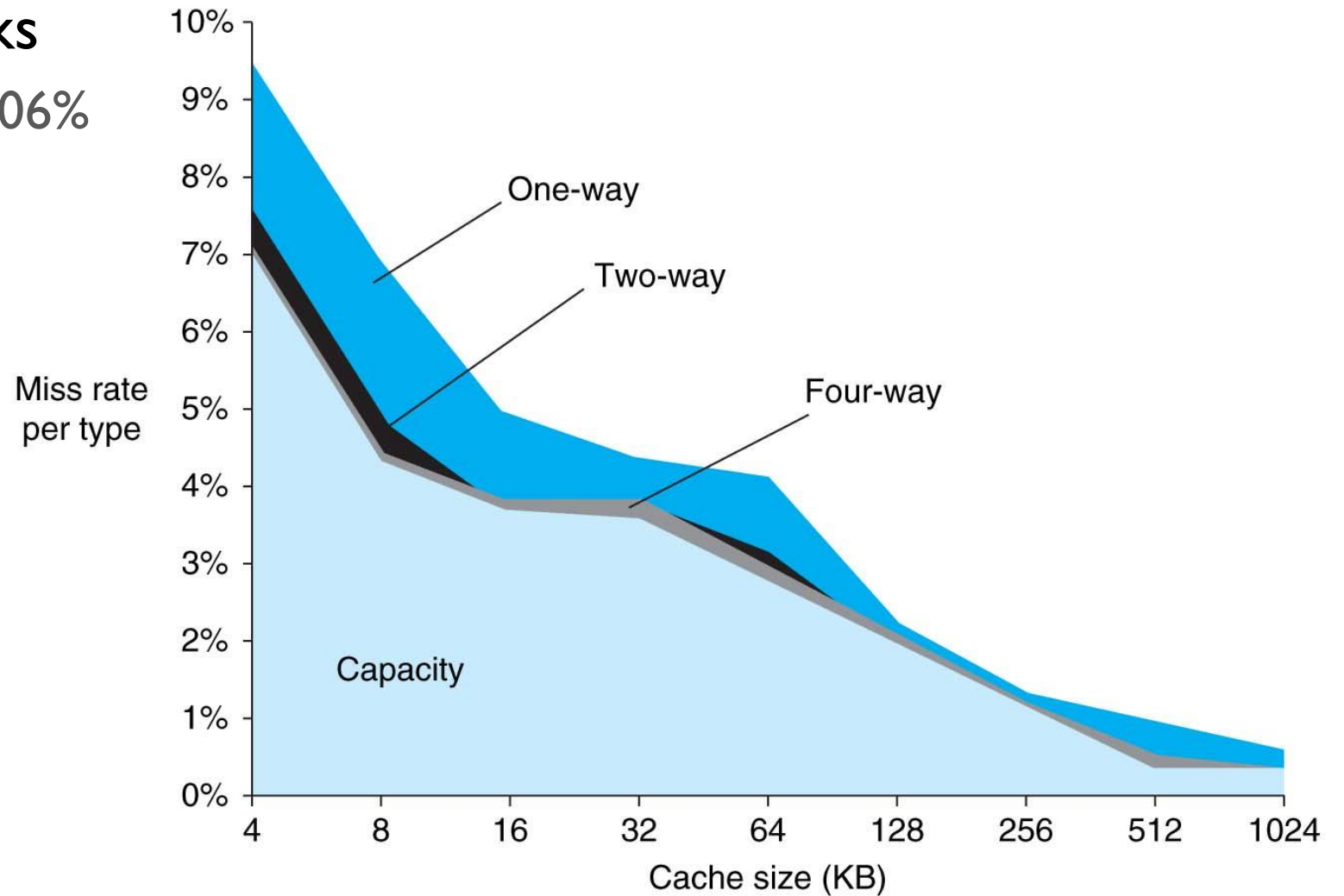
Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Multilevel On-Chip Caches

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

2-Level TLB Organization

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Summary

- **Fast memories are small, large memories are slow**
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺

- **Principle of locality**
  - Programs use a small part of their memory space frequently

- **Memory hierarchy**
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk

- **Memory system design is critical for multiprocessors**