Virtual Memory: Details

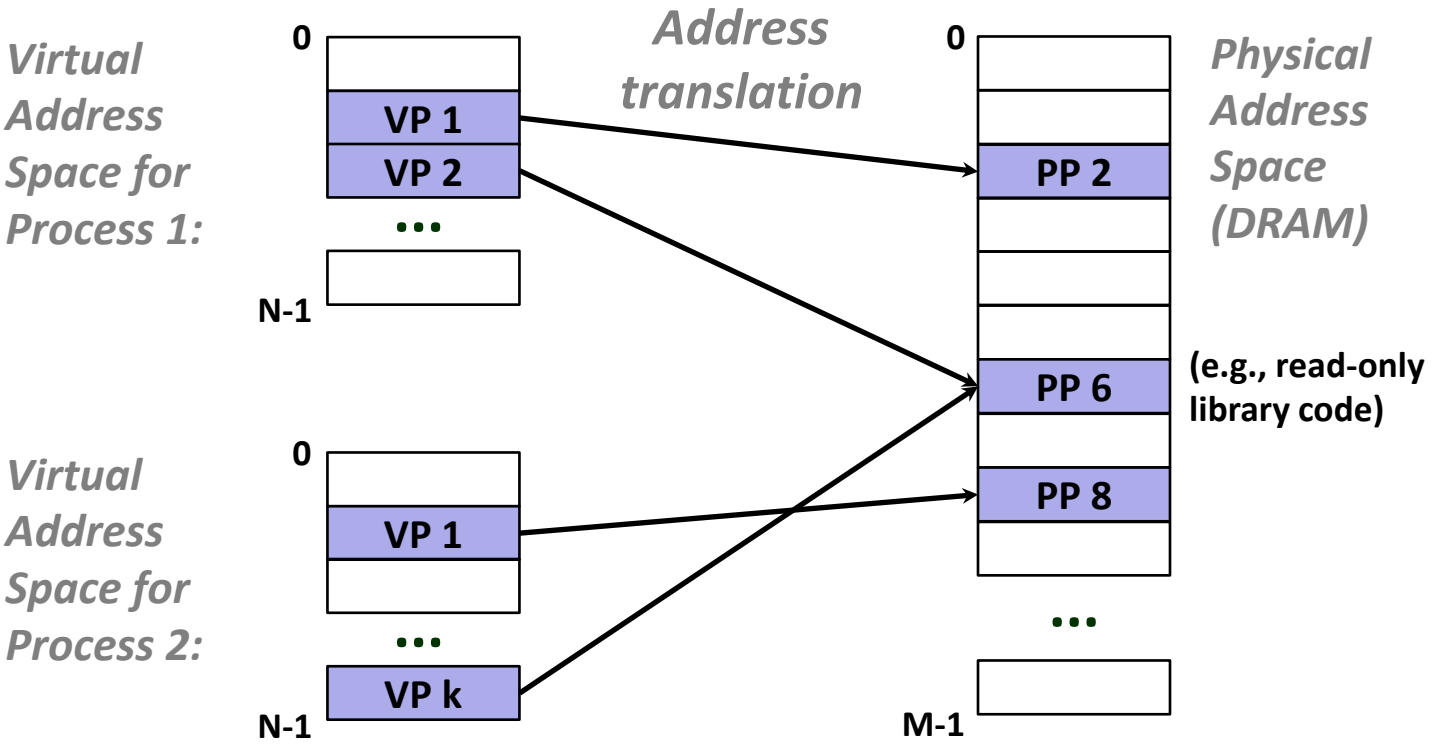
15-213/15-513: Introduction to Computer Systems
17th Lecture, June 14, 2024

Instructors:

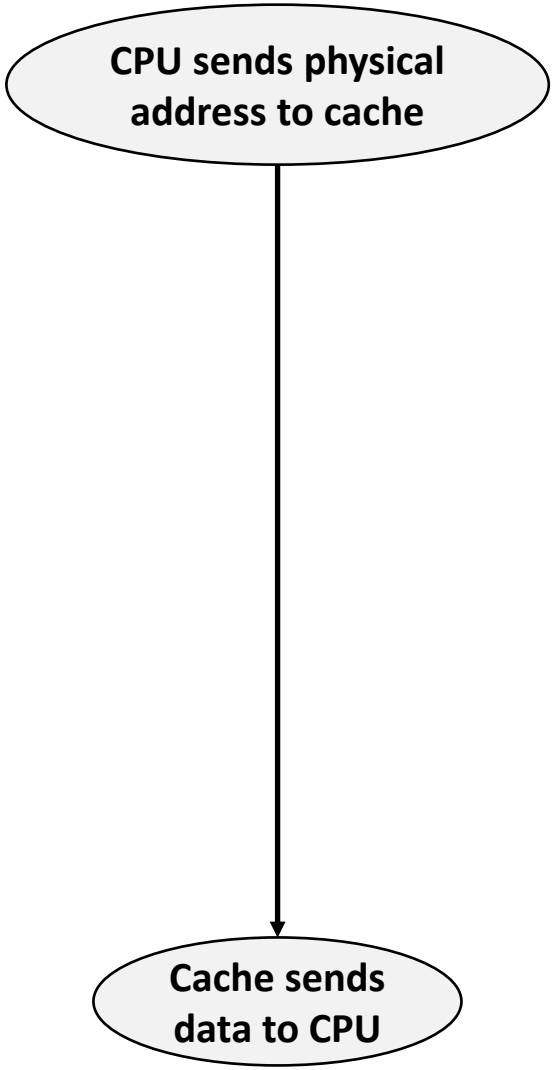
Brian Railing

Review: Virtual Addressing

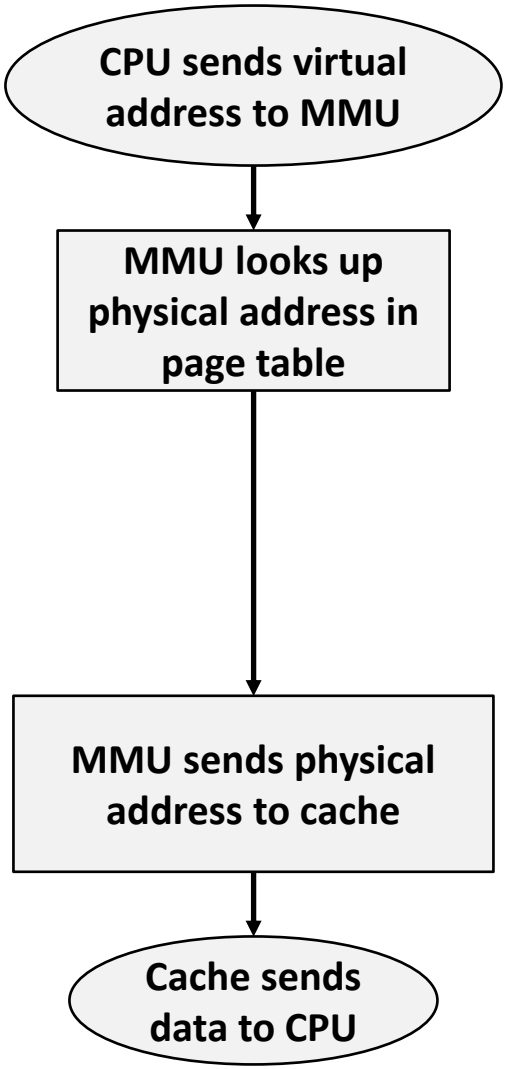
- Each process has its own *virtual address space*
- *Page tables* map virtual to physical addresses
- Physical memory can be shared among processes



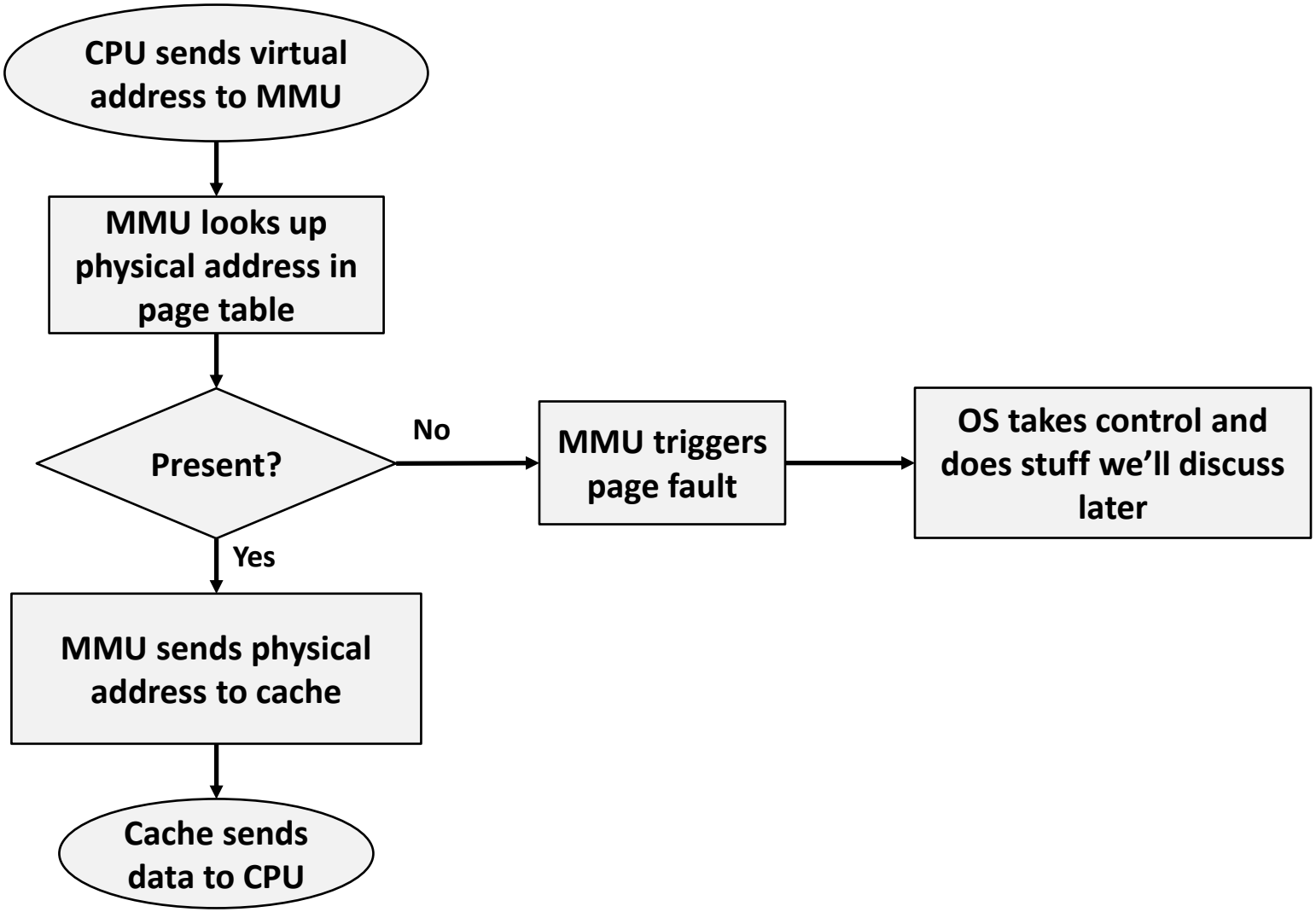
Review: Memory Accesses without VM



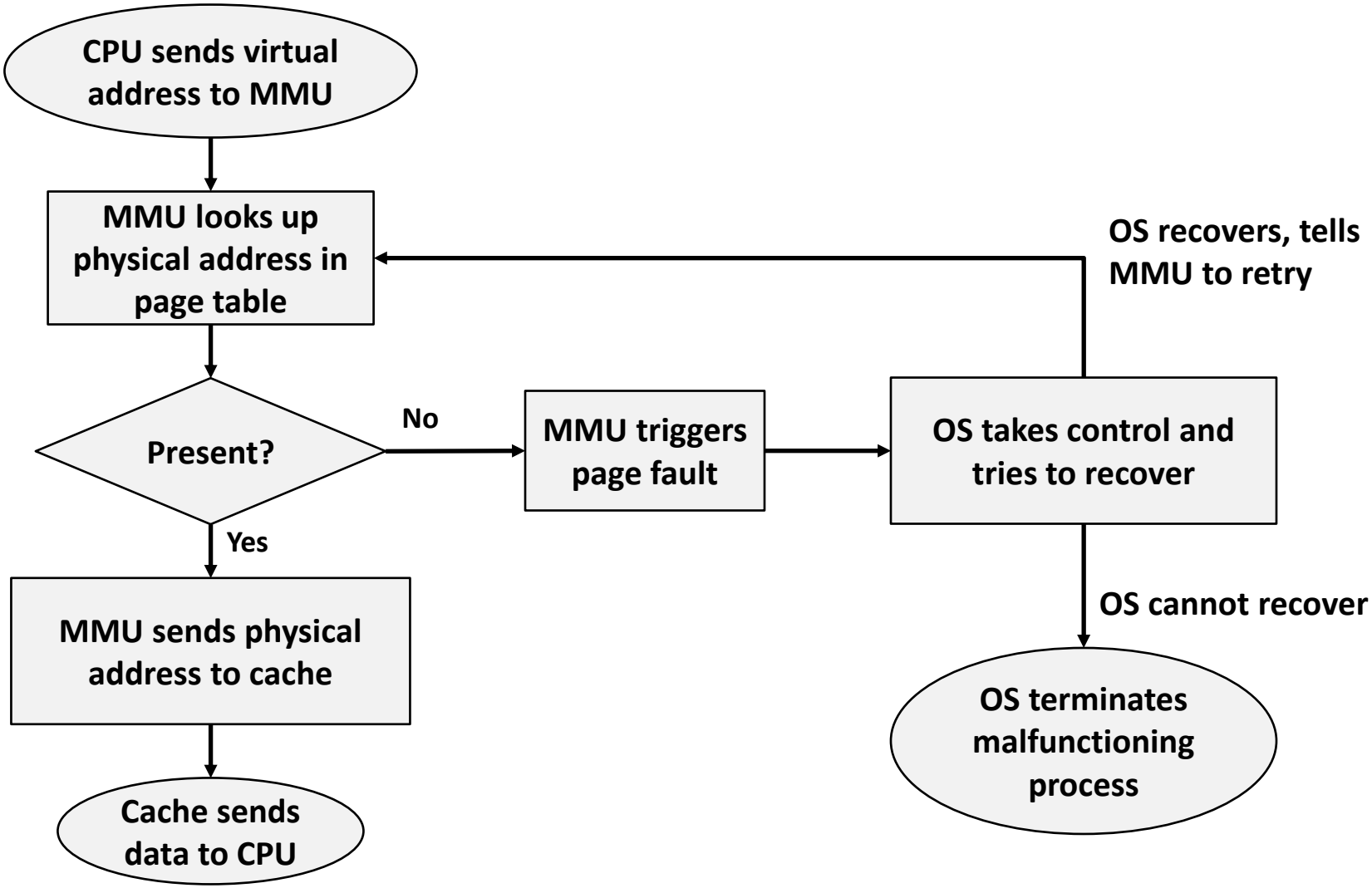
Review: Memory Accesses with VM



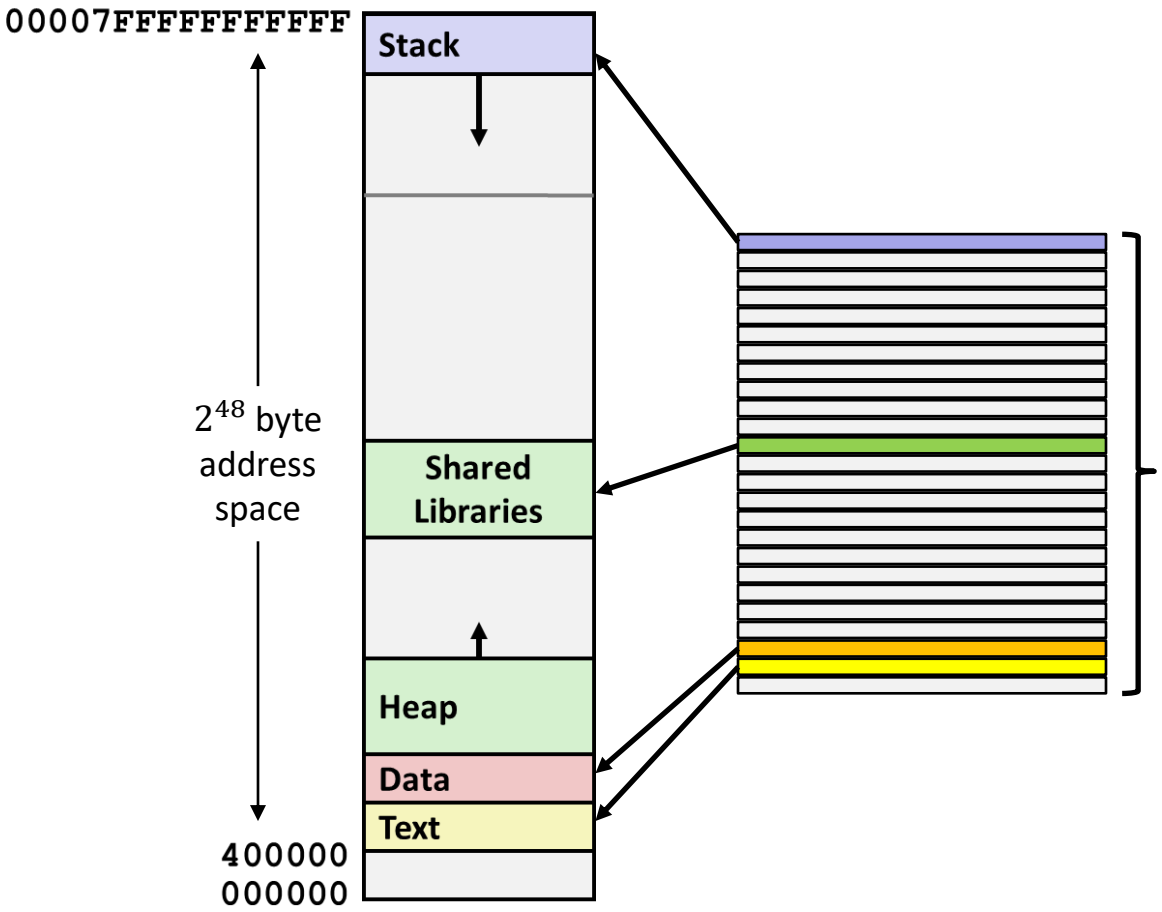
Review: Memory Accesses with VM



Review: Memory Accesses with VM



Review: The problem



One 64-bit array element for each 4096-byte page

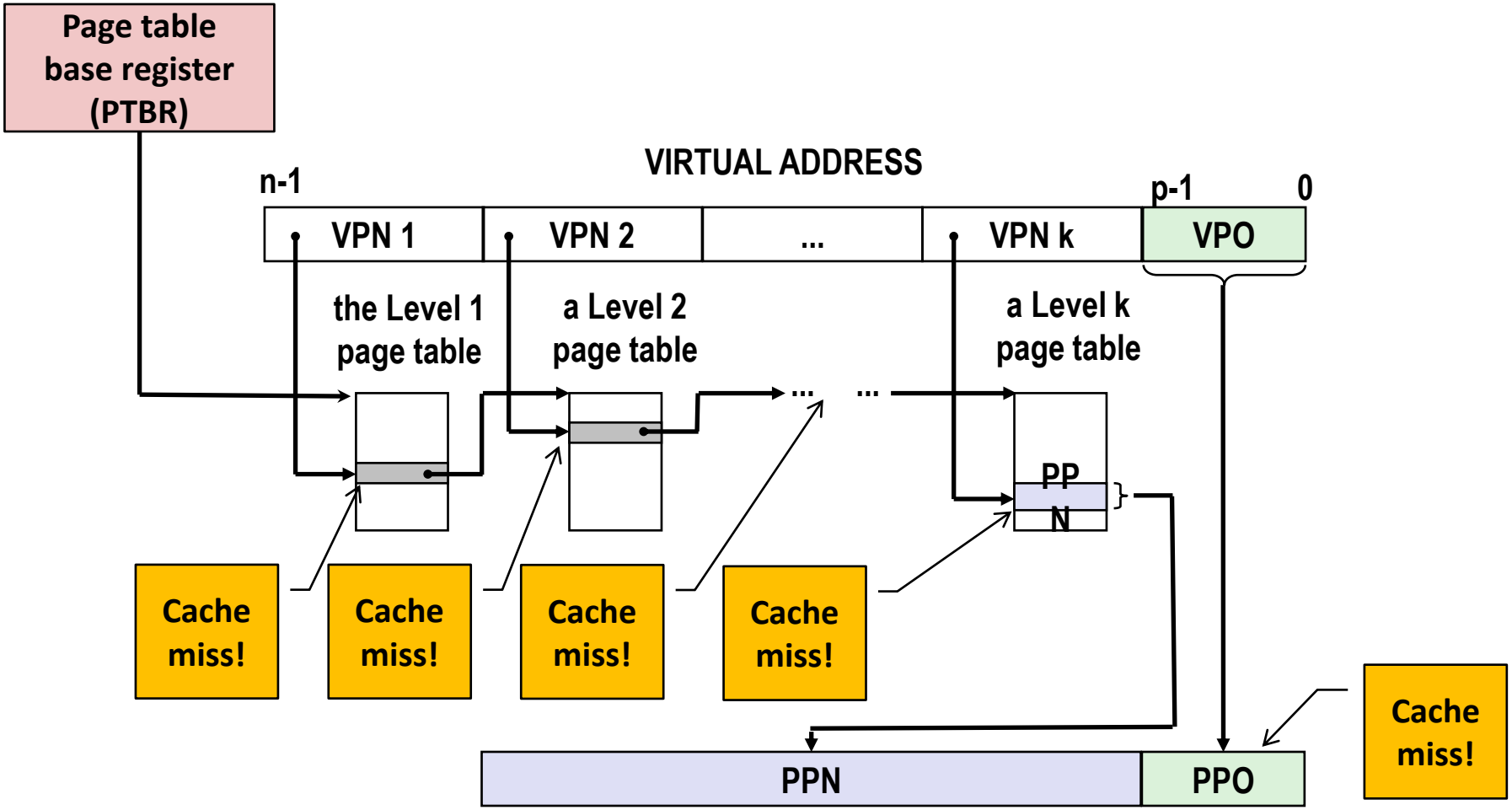
= $2^{48}/_{4096} \cdot 8$ bytes

= 2^{39} bytes

= 512 **gigabytes**

for **one** page table

Review: the problem (with k-level page tables)



Today

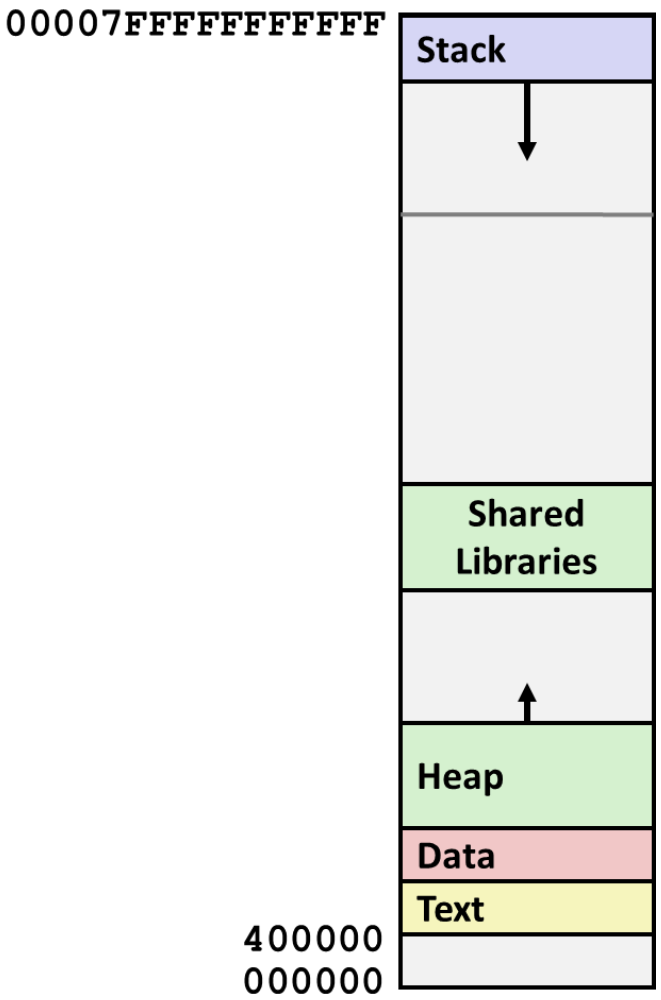
- **Page Faults**
- **Conceptual Quiz**
- **Concrete examples of virtual memory systems**
  - "Simple memory system" from CSAPP 9.6.4
  - Intel Core i7
- **Nifty things virtual memory makes possible**
  - Paging/swapping (disk as extra RAM)
  - Memory-mapped files (RAM as cache for disk)
  - Copy-on-write sharing

What happens on a page fault?

- What is a page fault?

- Why can they happen? How does the OS know?

OS tracks VM “areas”



- Each area has permissions / purpose / etc

Types of Faults

1. **This is a legal address**
   - Hard / Major faults – “normal” page faults
   - Soft / minor faults – the OS took the page away, but has not reused it

2. **The address is legal, but …**

   The type of access is wrong, so “protection exception”

3. **The address is not legal**

   Segmentation fault or bus error

Today

- Page Faults
- **Conceptual Quiz**
- Concrete examples of virtual memory systems
  - "Simple memory system" from CSAPP 9.6.4
  - Intel Core i7
- Nifty things virtual memory makes possible
  - Paging/swapping (disk as extra RAM)
  - Memory-mapped files (RAM as cache for disk)
  - Copy-on-write sharing

Conceptual Quiz: 1

For a simple system with a one-level page table, what must the MMU do before fetching the PTE?

Before fetching the Page Table Entry (PTE) from a one-level page table in a simple system, the Memory Management Unit (MMU) performs the following sub-steps:

1. **Virtual Address Splitting**: The MMU splits the virtual address into two parts:
    1. **Virtual Page Number (VPN)**: The leftmost bits of the address represent the VPN.
    2. **Virtual Page Offset (VPO)**: The remaining bits indicate the offset within the page.
2. **Indexing into Page Table**: The VPN is used directly as an array index in the page table. The MMU looks up the corresponding PTE using the VPN.

Feel free to ask if you need further clarification! 😊

For additional context and details, you can refer to the lecture materials from Carnegie Mellon University[12]

Conceptual Quiz: 2

The MMU must know the *physical* address of the page table in order to read page table entries from memory. Why does it need a physical address?

The Memory Management Unit (MMU) requires the **physical address** of the page table for several reasons:

Address Translation: The MMU translates **virtual addresses** (used by the CPU) to **physical addresses** (used by RAM). To do this, it needs to access the page table.

Page Table Lookup: When a virtual address is accessed, the MMU uses the VPN (Virtual Page Number) to index into the page table. The resulting PTE (Page Table Entry) contains the physical address of the corresponding page frame in RAM.

Hardware Interaction: The MMU interacts directly with RAM and other hardware components. It needs the physical address to perform memory accesses efficiently.

Protection and Permissions: The page table entries also store information about page permissions (read-only, read-write, etc.) and protection flags. The MMU checks these flags to enforce memory protection.

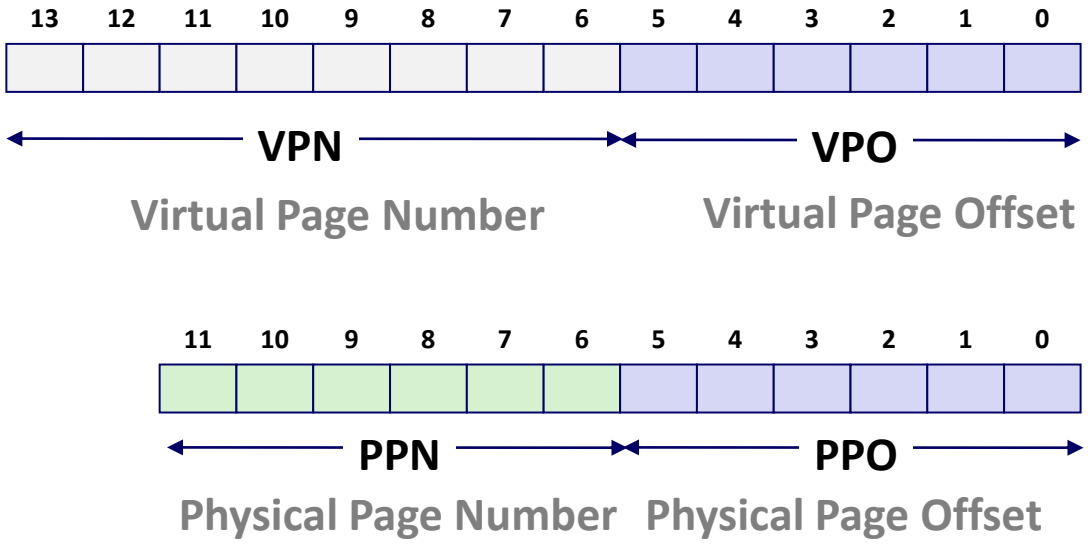
In summary, the physical address of the page table allows the MMU to perform address translation, look up PTEs, and manage memory efficiently. If you have further questions, feel free to ask!

Today

- Page Faults
- Conceptual Quiz
- **Concrete examples of virtual memory systems**
  - "Simple memory system" from CSAPP 9.6.4
  - Intel Core i7
- Nifty things virtual memory makes possible
  - Paging/swapping (disk as extra RAM)
  - Memory-mapped files (RAM as cache for disk)
  - Copy-on-write sharing

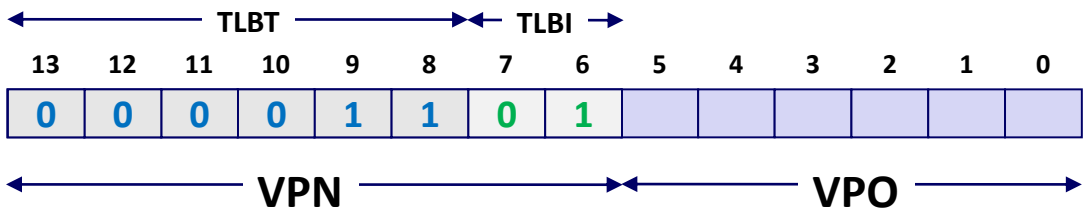
Simple Memory System Example

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



Simple Memory System TLB

- 16 entries
- 4-way associative



VPN = 0b1101 = 0x0D

Translation Lookaside Buffer (TLB)

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

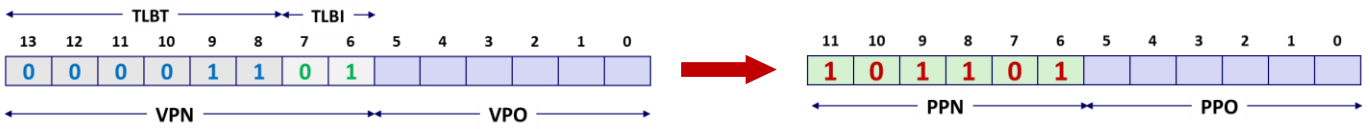
Simple Memory System Page Table

- Only showing the 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

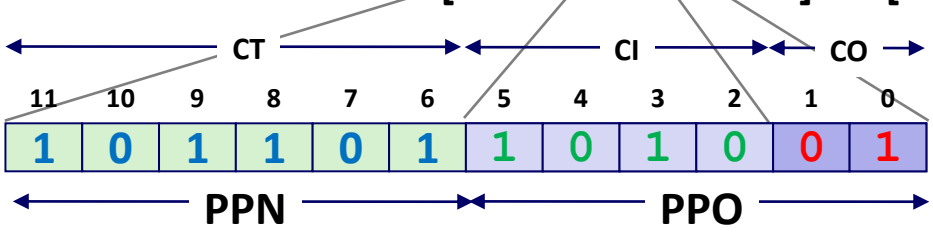
0x0D → 0x2D



Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped

V[0b0000110110100 1] = V[0x369]
P[0b1011011010 01] = P[0xB69] = 0x15

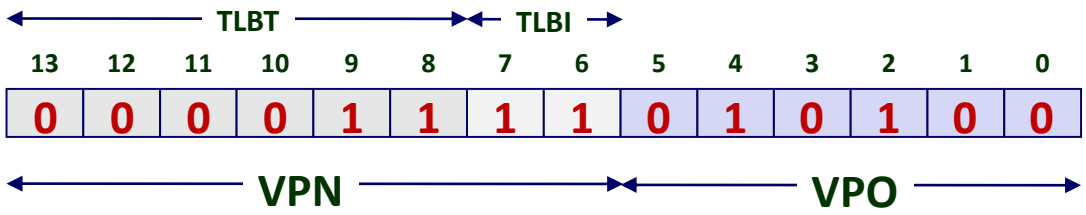


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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

Virtual Address: 0x03D4

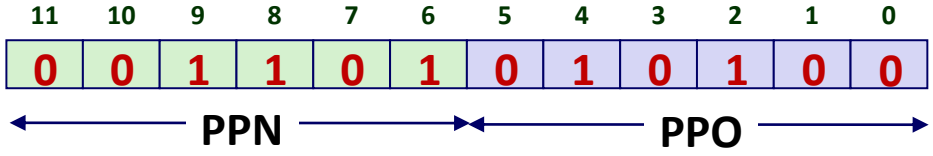


VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

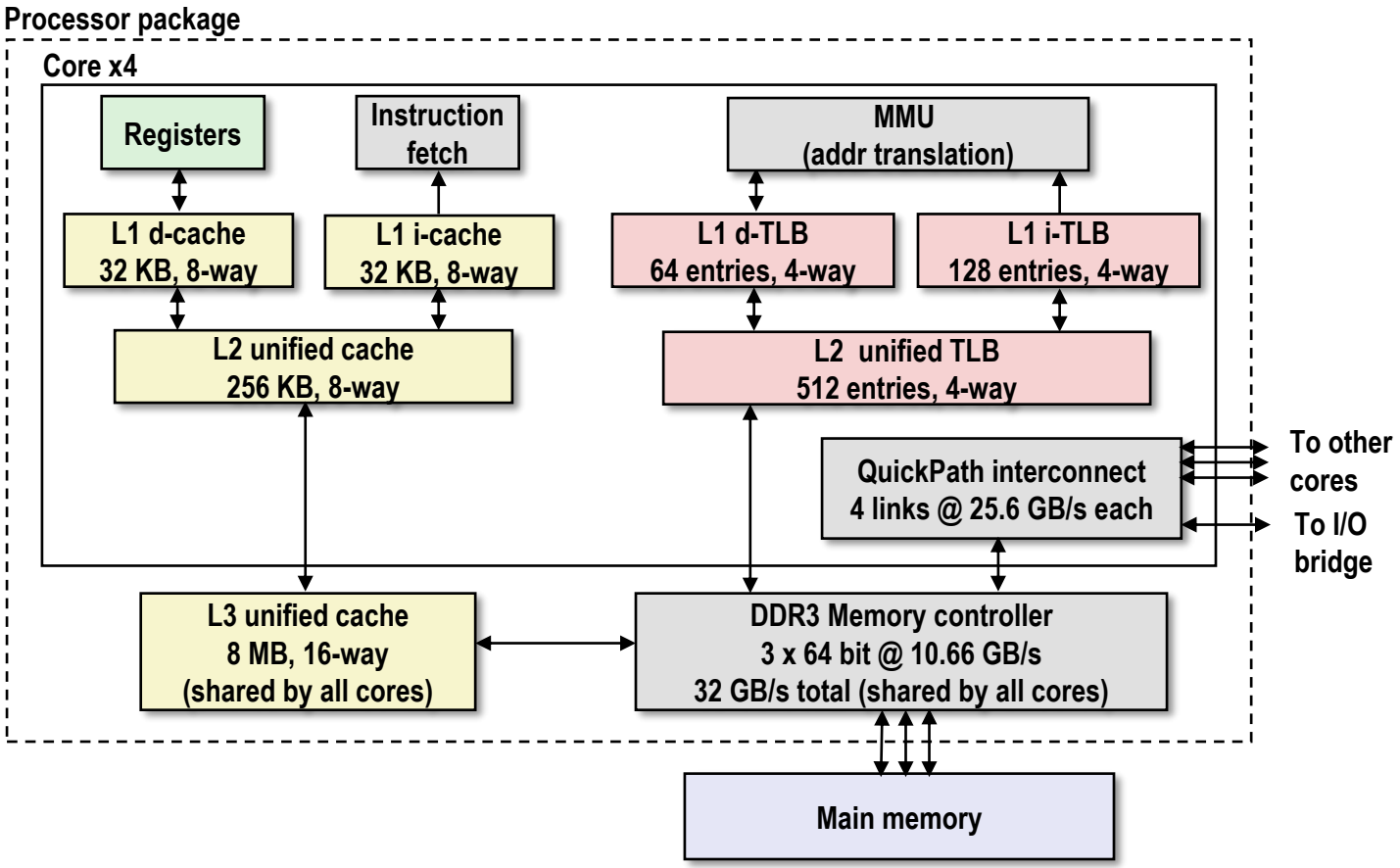
TLB

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

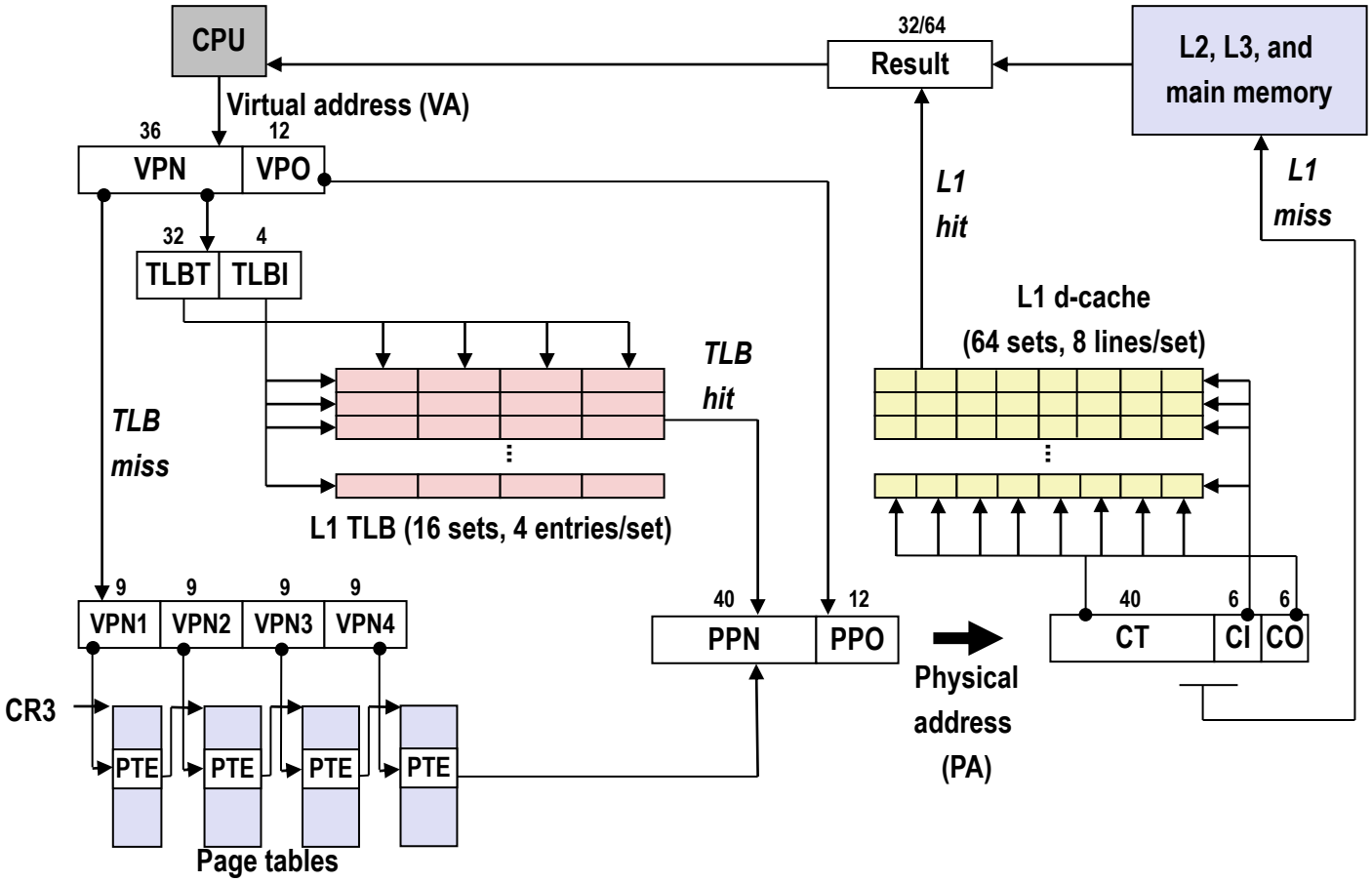
Physical Address



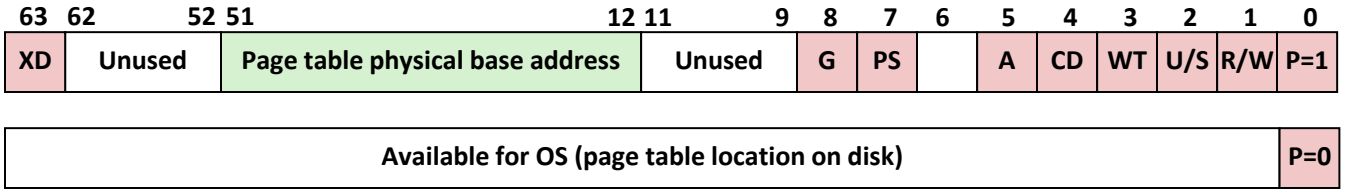
Intel Core i7 Memory System



End-to-end Core i7 Address Translation



Core i7 Level 1-3 Page Table Entries



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.

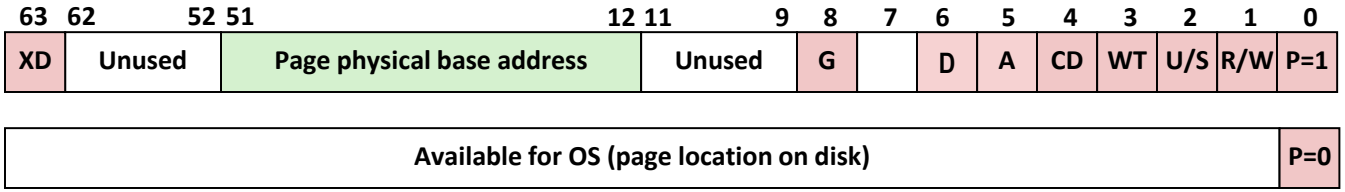
A: Reference bit (set by MMU on reads and writes, cleared by software).

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.

Core i7 Level 4 Page Table Entries



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)

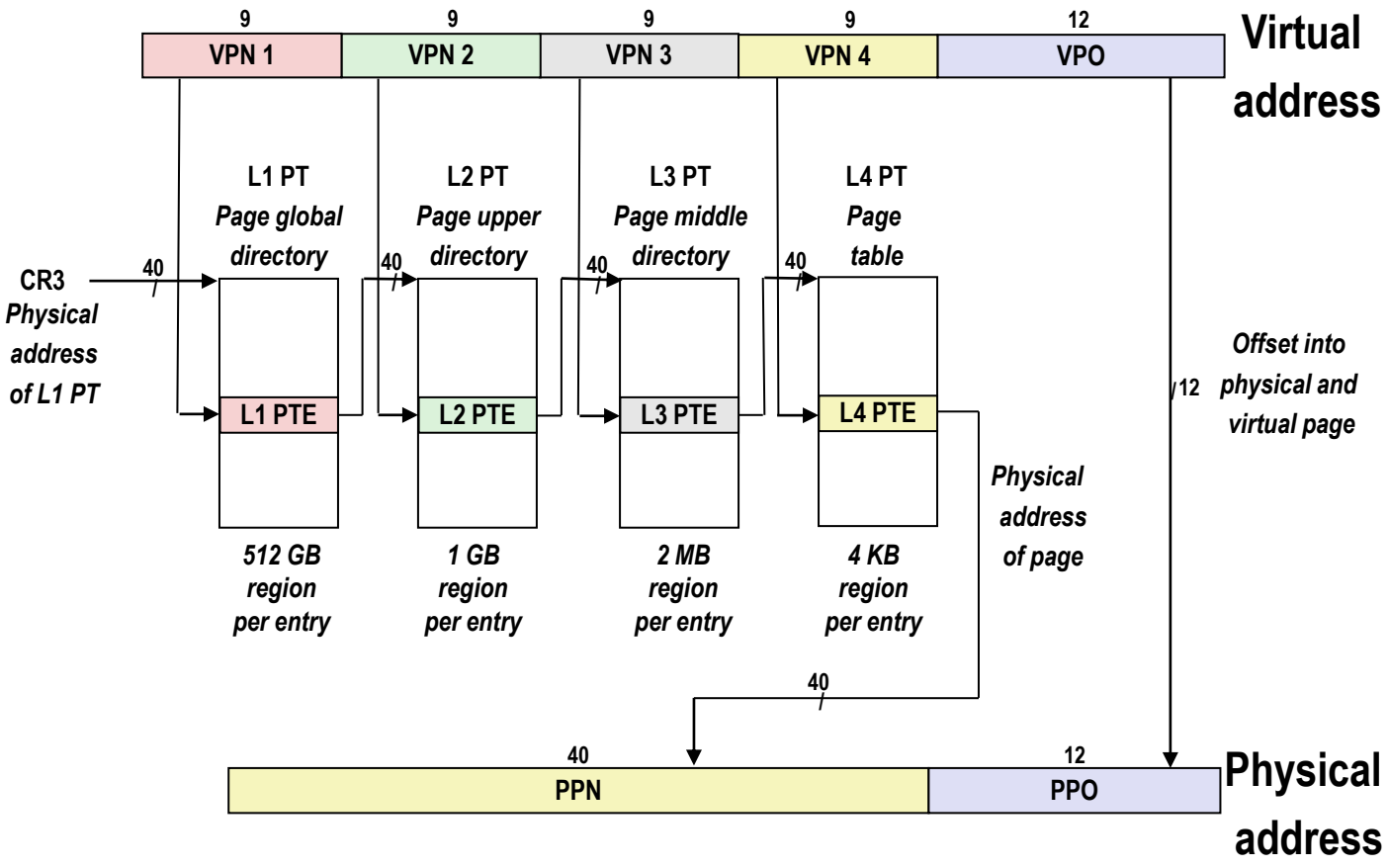
D: Dirty bit (set by MMU on writes, cleared by software)

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

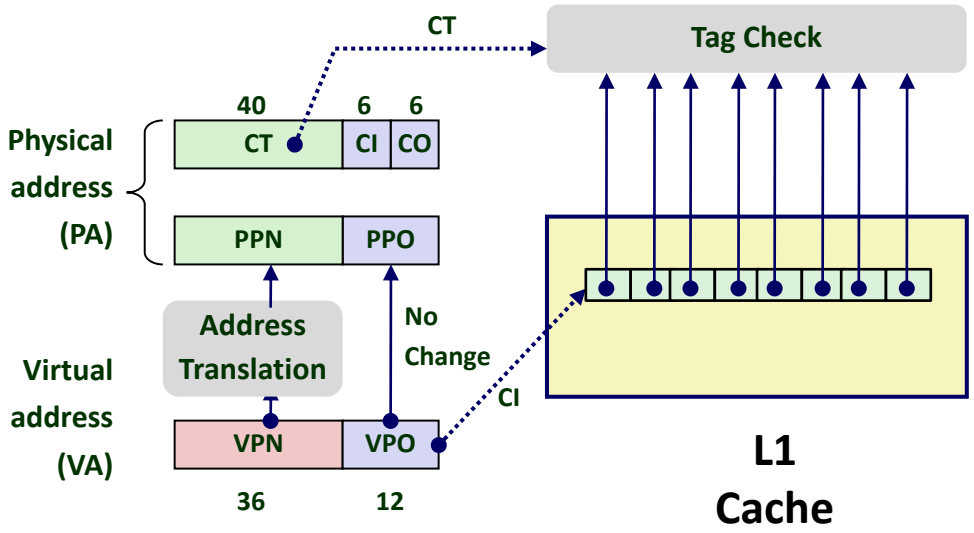
Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

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 quickly
  - "Virtually indexed, physically tagged"
  - Cache carefully sized to make this possible

Today

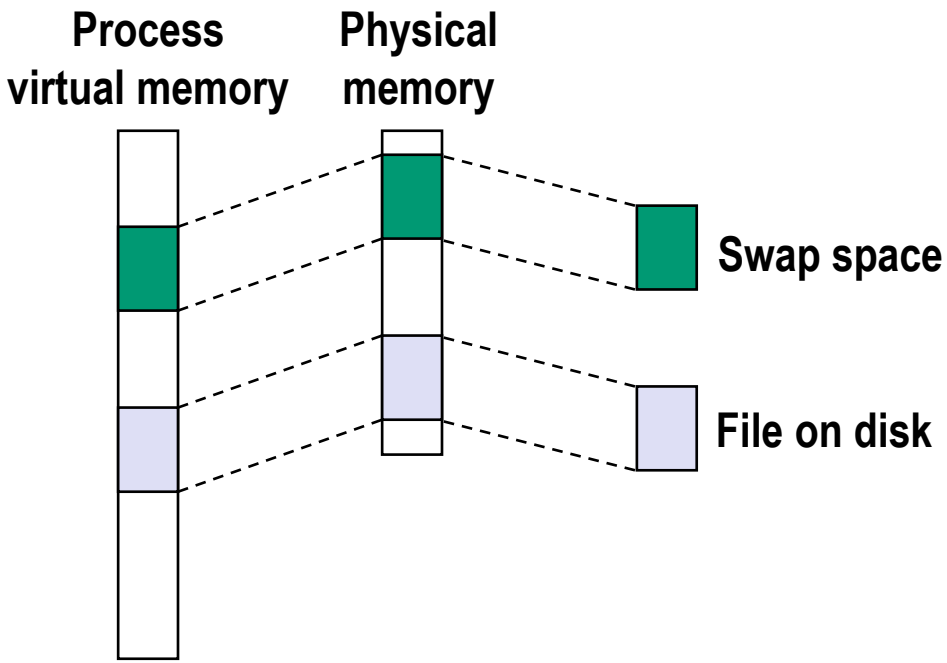
- Page Faults
- Conceptual Quiz
- Concrete examples of virtual memory systems
  - "Simple memory system" from CSAPP 9.6.4
  - Intel Core i7
- **Nifty things virtual memory makes possible**
  - Memory-mapped files (RAM as cache for disk)
  - Copy-on-write sharing

Memory-Mapped Files

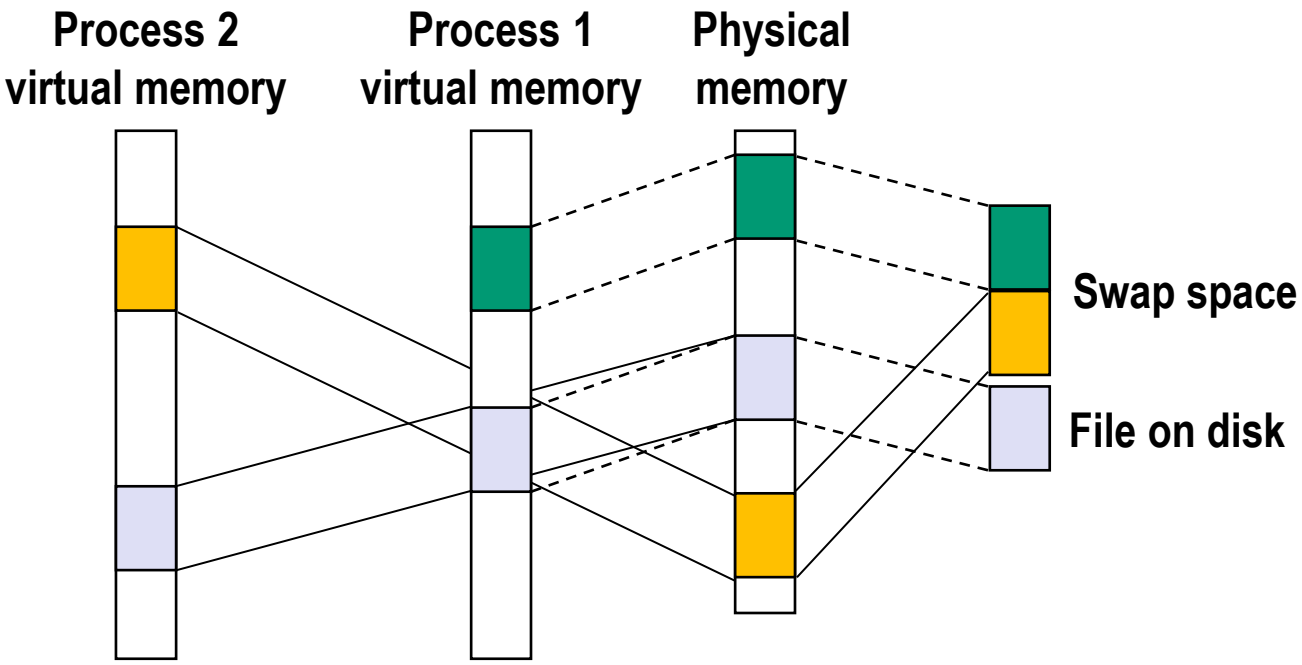
- **Paging = every page of a program's physical RAM is *backed* by some page of disk***
- **Normally, those pages belong to *swap space***
- **But what if some pages were backed by … files?**

* This is how it used to work 20 years ago.
Nowadays, not always true.

Memory-Mapped Files



Memory-Mapped Files

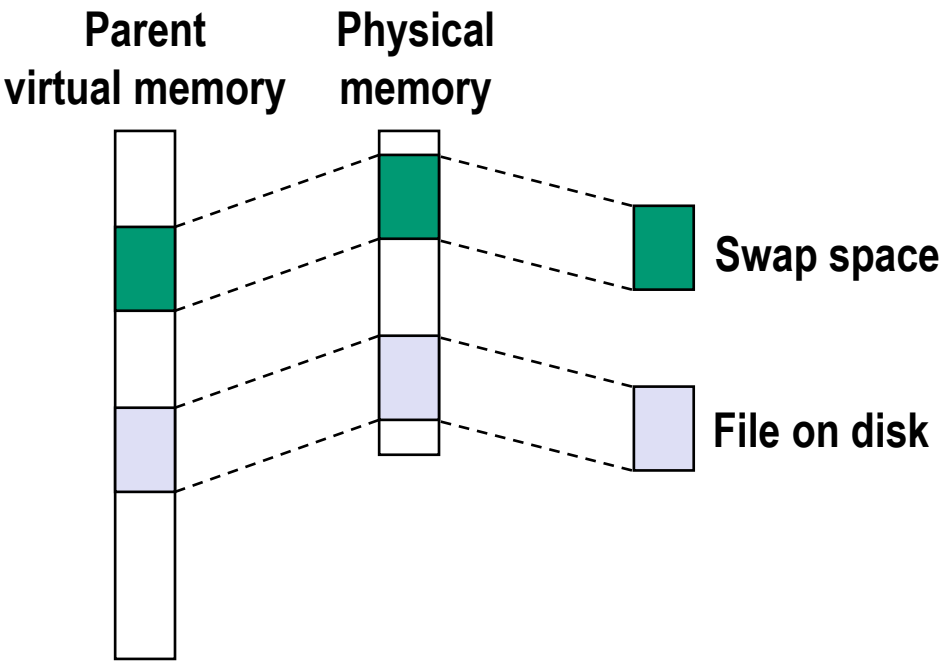


Demo

Show mmap from sfs

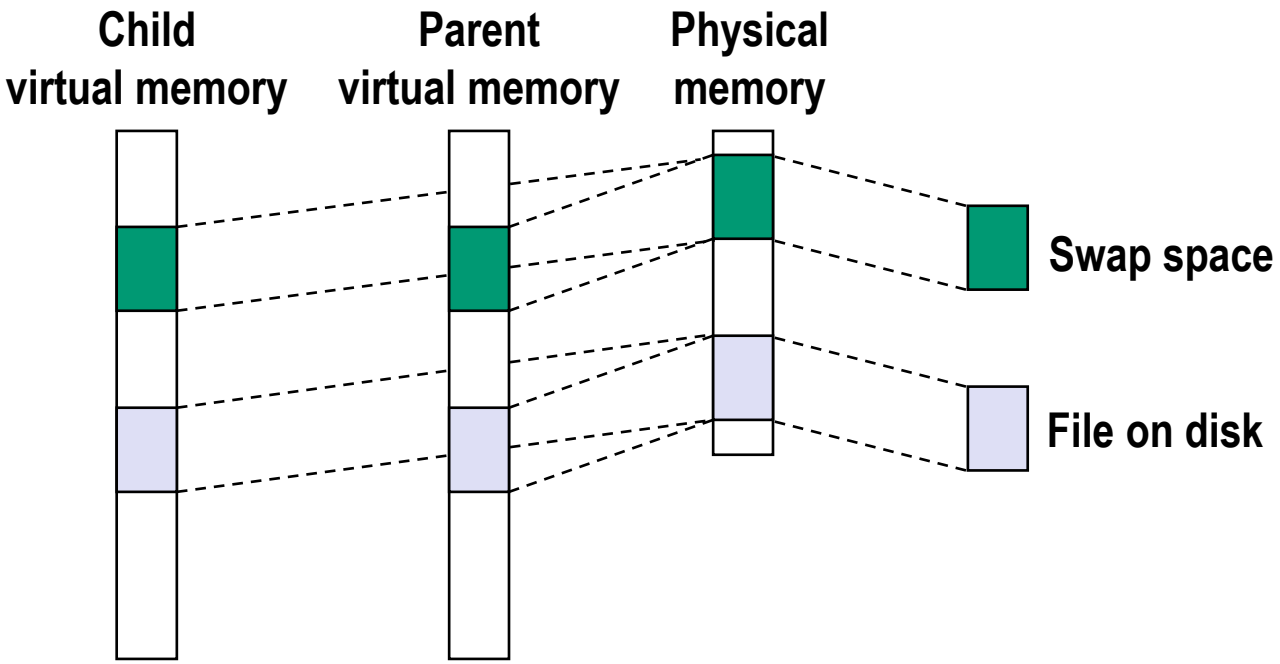
Copy-on-write sharing

- **`fork` creates a new process by copying the entire address space of the parent process**
  - That sounds slow
  - It *is* slow



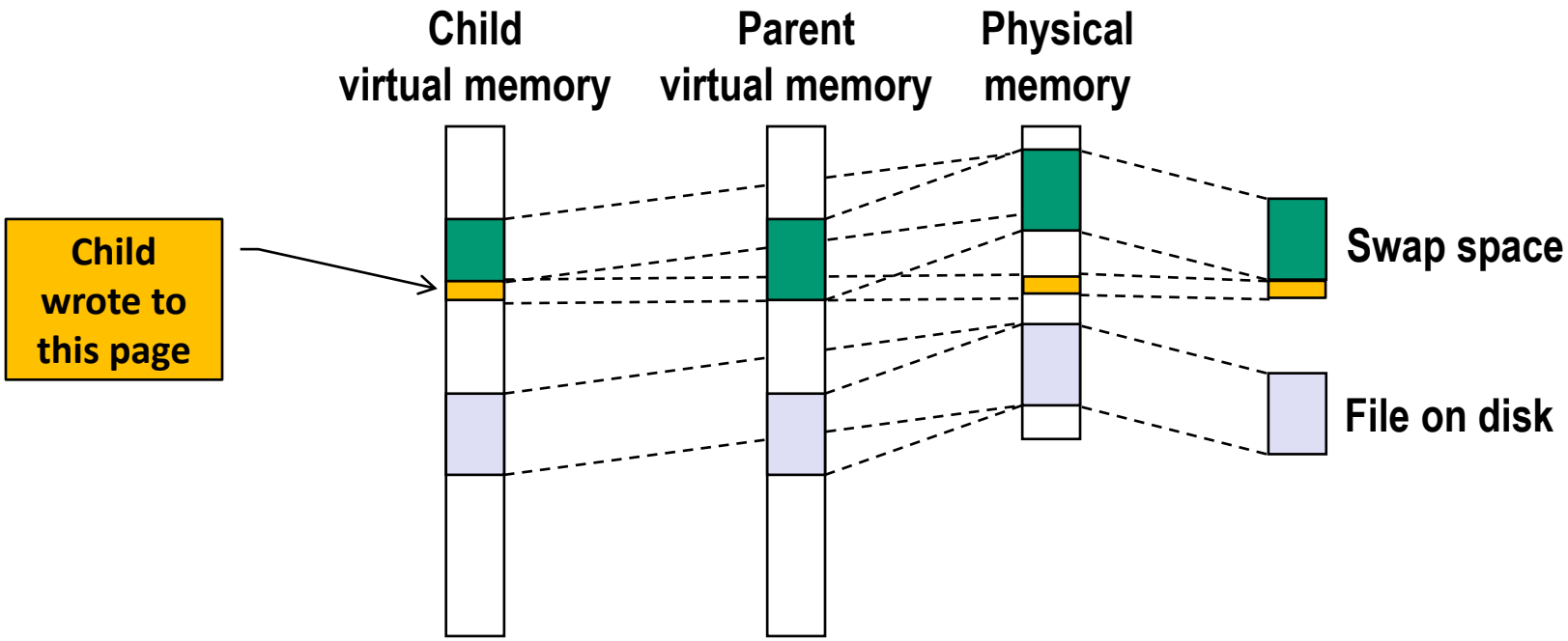
- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Copy-on-write sharing



- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Copy-on-write sharing



- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Summary

- **Multi-level page tables reduce total memory consumption of page tables**
- **Translation lookaside buffers reduce time cost of translation**
- **Real systems have 3 to 5 levels of page table**
- **Virtual memory makes nifty things possible**
  - Memory protection and process isolation
  - Paging/swapping (disk as extra RAM)
  - Memory-mapped files (RAM as cache for disk)
  - Copy-on-write sharing

Conceptual Quiz: 3

Why are one-level page tables impractical and how do multi-level page tables fix this problem?

A single-level page table covering the entire address space of a typical system would be much too large. For instance, with 4kB pages, a 48-bit address space, and a 8-byte PTE, a single-level page table would occupy 512 *gigabytes*, which is more RAM than most computers have.

Conceptual Quiz: 4

Why is memory access slower with a multi-level page table than with a single-level page table?

A *k*-level page table requires *k* memory loads in order to determine the physical address. There is no spatial locality to these loads.

Conceptual Quiz: 5

What is the Translation Lookaside Buffer (TLB), what problem does it solve, and when is it used?

The TLB is a small cache dedicated to storing mappings from virtual to physical addresses. It avoids the cost of lookups in a multi-level page table.

The MMU consults the TLB for each address as its first action; if there is a TLB hit, it does not need to fetch anything from the page table.

Conceptual Quiz: 6

How does virtual memory interact with the memory cache(s)?

The cache's function is to speed up access to whatever data is most frequently used. The MMU sits "in between" the CPU and the cache; the cache works only with physical addresses. This means data from multiple processes may coexist in the cache (or compete for cache space).