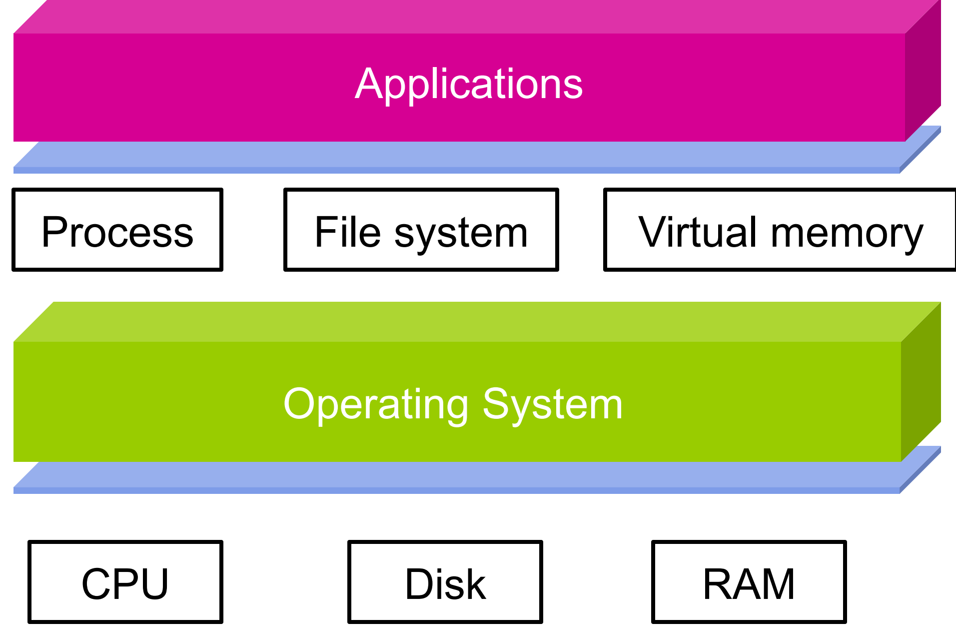
Advanced Operating Systems (CS 202)

Virtual Memory

OS Abstractions



An Example Memory Hierarchy

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

- L0: Registers — CPU registers hold words retrieved from L1 cache
- L1: L1 cache (SRAM) — L1 cache holds cache lines retrieved from L2 cache
- L2: L2 cache (SRAM) — L2 cache holds cache lines retrieved from main memory
- L3: Main memory (DRAM) — Main memory holds disk blocks retrieved from local disks
- L4: Local secondary storage (local disks) — Local disks hold files retrieved from disks on remote network servers
- L5: Remote secondary storage (tapes, distributed file systems, Web servers)

Memory hierarchy

- *Cache:* A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

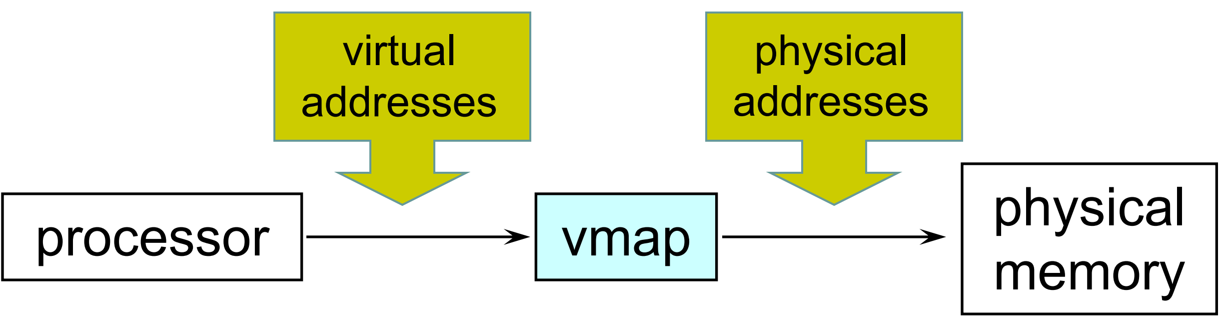
- Fundamental idea of a memory hierarchy:
  - For each layer, faster, smaller device caches larger, slower device

.

- Why do memory hierarchies work?
  - Because of locality!
    - Hit fast memory much more frequently even though its smaller
  - Thus, the storage at level k+1 can be slower (but larger and cheaper!)

- *Big Idea:* The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

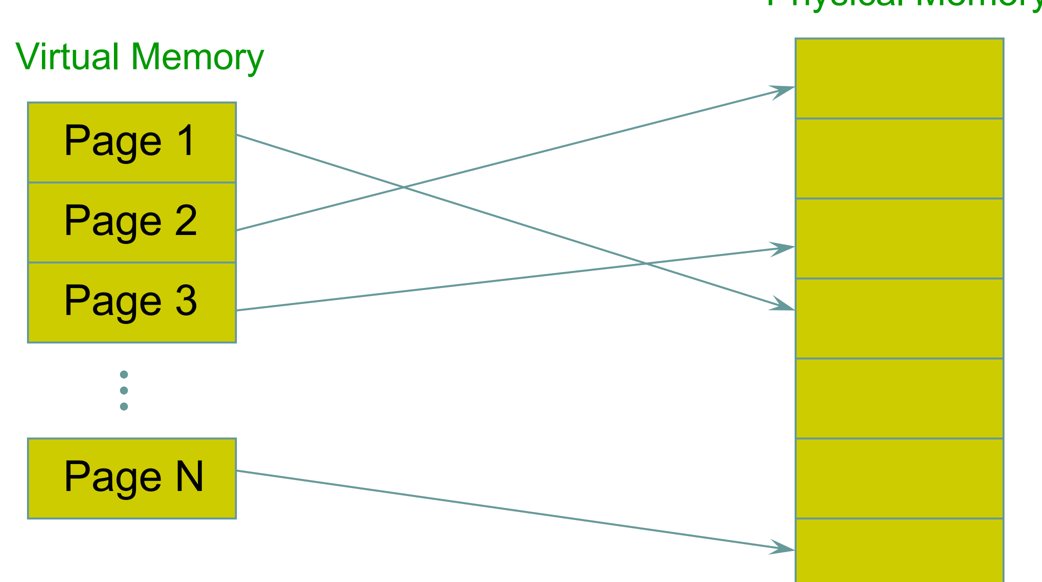
Virtual Addresses



- Many ways to do this translation…
  - Need hardware support and OS management algorithms
- Requirements
  - Need protection – restrict which addresses jobs can use
  - Fast translation – lookups need to be fast
  - Fast change – updating memory hardware on context switch

Paging

- Main Idea: split virtual address space into multiple partitions
  - Each can go anywhere!



Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory

But need to keep track of where things are!

Page Lookups

Physical Memory

Virtual Address

Page number	Offset

Page Table

Page frame

Physical Address

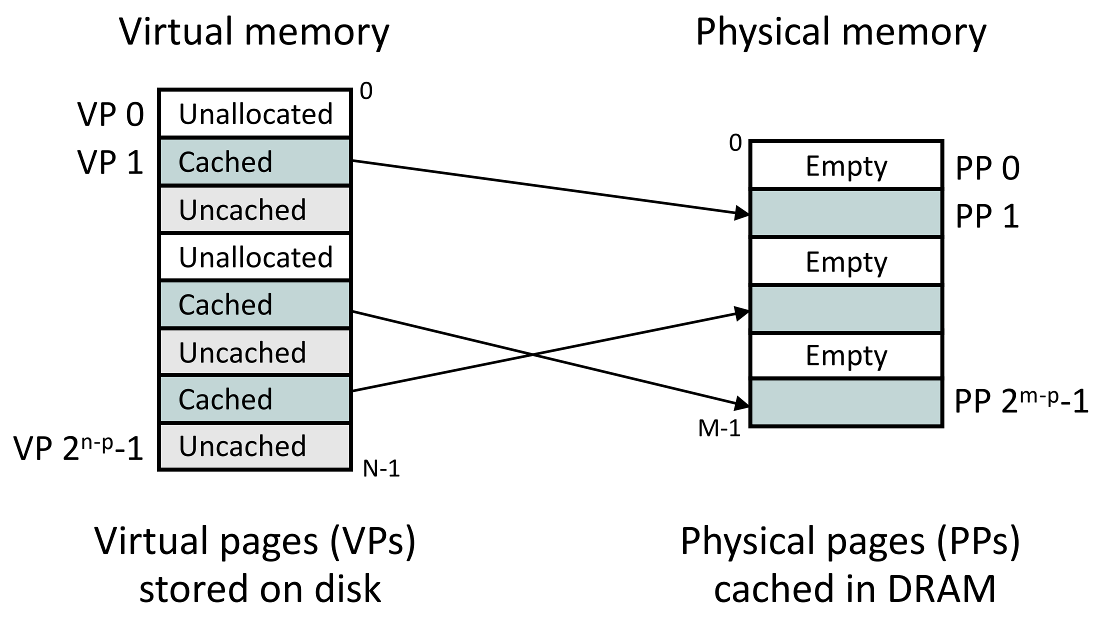
Page frame	Offset

Why Virtual Memory (VM)?

- Virtual memory is page with a new ingredient
  - Allow pages to be on disk
    - In a special partition (or file) called swap
- Motivation?
  - Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
  - With VM, this can be big!

VM as a Tool for Caching

- *Virtual memory* is an array of N contiguous bytes stored on disk.
- The contents of the array on disk are cached in *physical memory* (*DRAM cache*)
  - These cache blocks are called *pages* (size is $P = 2^p$ bytes)



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

Physical memory (DRAM)

Physical page number or disk address

Valid

	Valid	Physical page number or disk address
PTE 0	0	null
	1	
	1	
	0	
	1	
	0	null
	0	
PTE 7	1	

Memory resident page table (DRAM)

Physical memory (DRAM)	
VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

Virtual memory (disk)

VP 1

VP 2

VP 3

VP 4

VP 6

VP 7

Page Hit

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

Page Fault

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

Handling Page Fault

- Page miss causes page fault (an exception)

Virtual address

Valid | *Physical page number or disk address*

	Valid	Physical page number or disk address
PTE 0	0	null
	1	•
	1	•
	0	•
	1	•
	0	null
	0	•
PTE 7	1	•

Memory resident page table (DRAM)

Physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

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)

Virtual address

Valid

Physical page number or disk address

	Valid	Physical page number or disk address
PTE 0	0	null
	1	•
	1	•
	1	•
	0	•
	0	null
	0	•
PTE 7	1	•

Memory resident page table (DRAM)

Physical memory (DRAM)

Physical memory	
VP 1	PP 0
VP 2	
VP 7	
VP 3	PP 3

Virtual memory (disk)

Virtual memory
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

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!

VM as a Tool for Mem 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

VM as a Tool for Mem 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

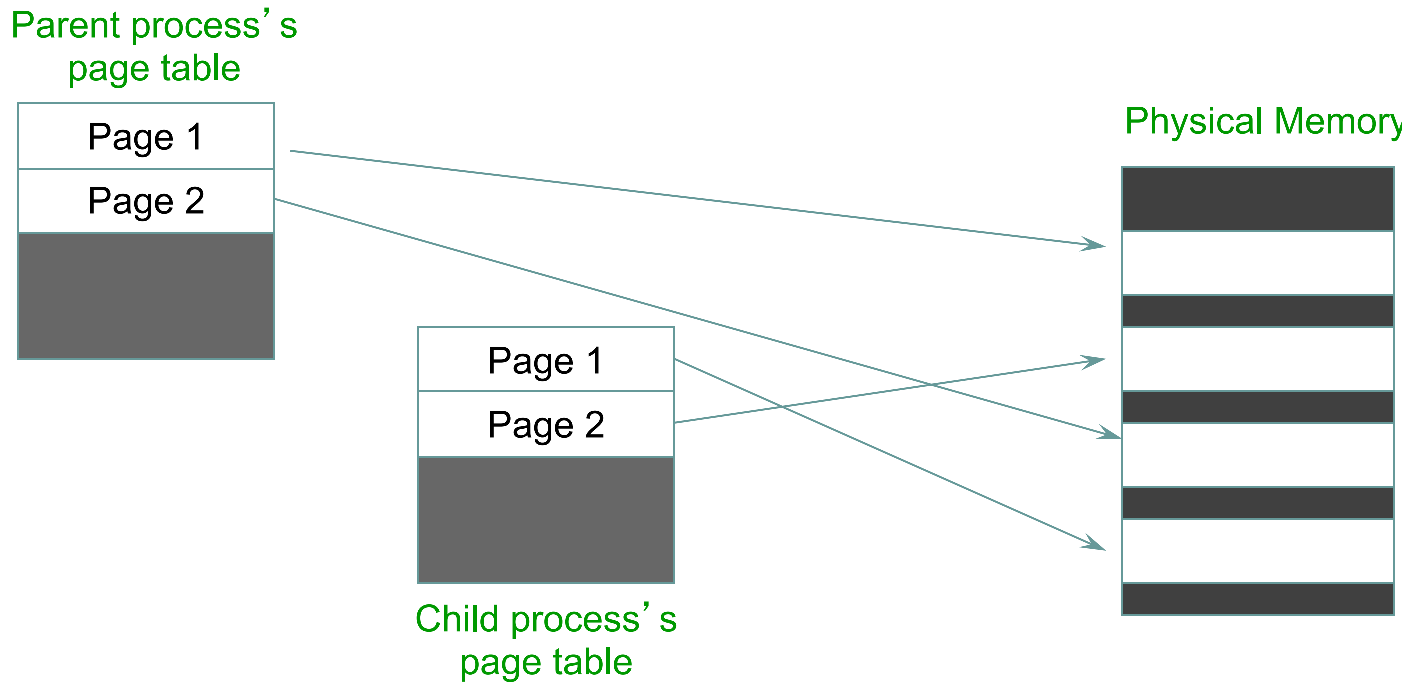
Sharing

- Can map shared memory at same or different virtual addresses in each process' address space
  - Different:
    - 10th virtual page in P1 and 7th virtual page in P2 correspond to the 2nd physical page
    - Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same:
    - 2nd physical page corresponds to the 10th virtual page in both P1 and P2
    - Less flexible, but shared pointers are valid

Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
  - How does this help fork()?

Execution of fork()



fork() with Copy on Write

When either process modifies Page 1, page fault handler allocates new page and updates PTE in child process

Parent process’s page table

Page 1
Page 2

Physical Memory

Page 1
Page 2

Child process’s page table

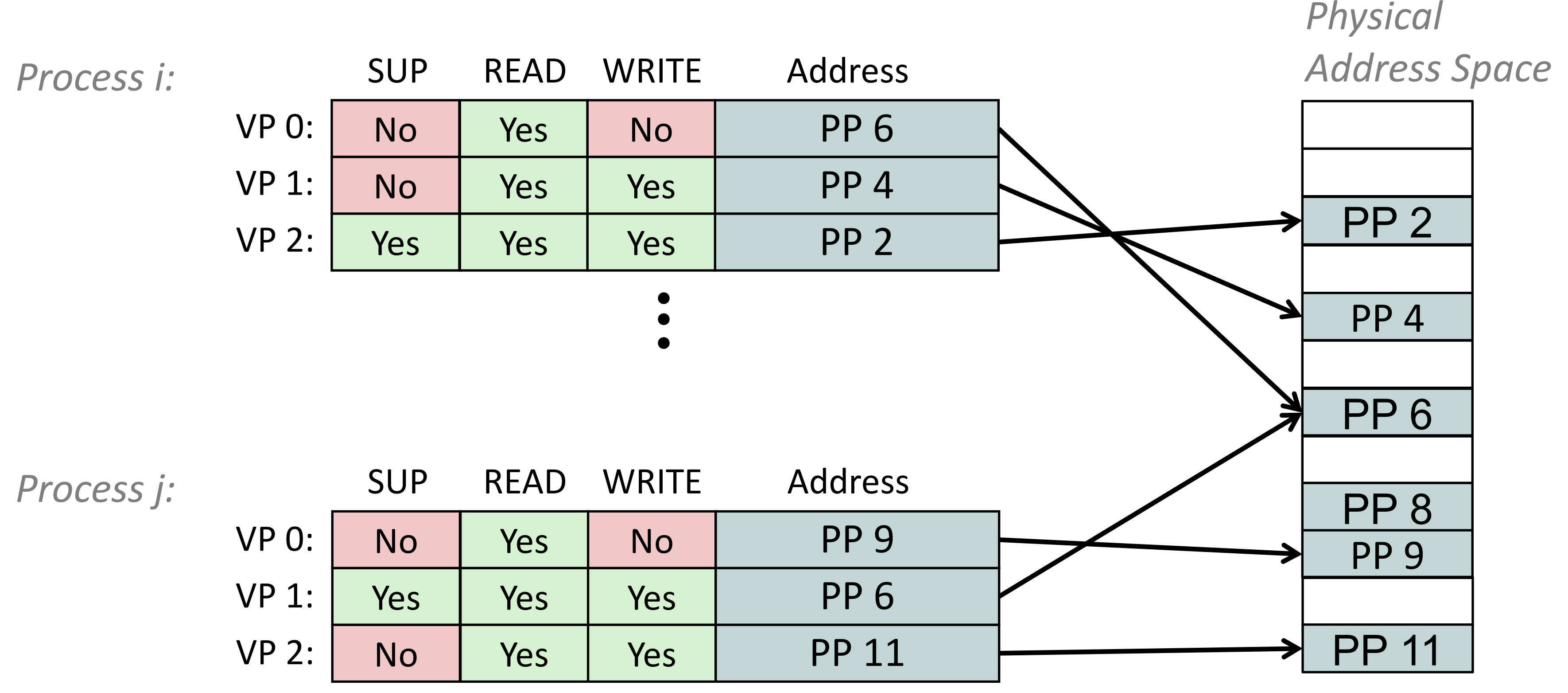
Protection bits set to prevent either process from writing to any page

Simplifying Linking and Loading

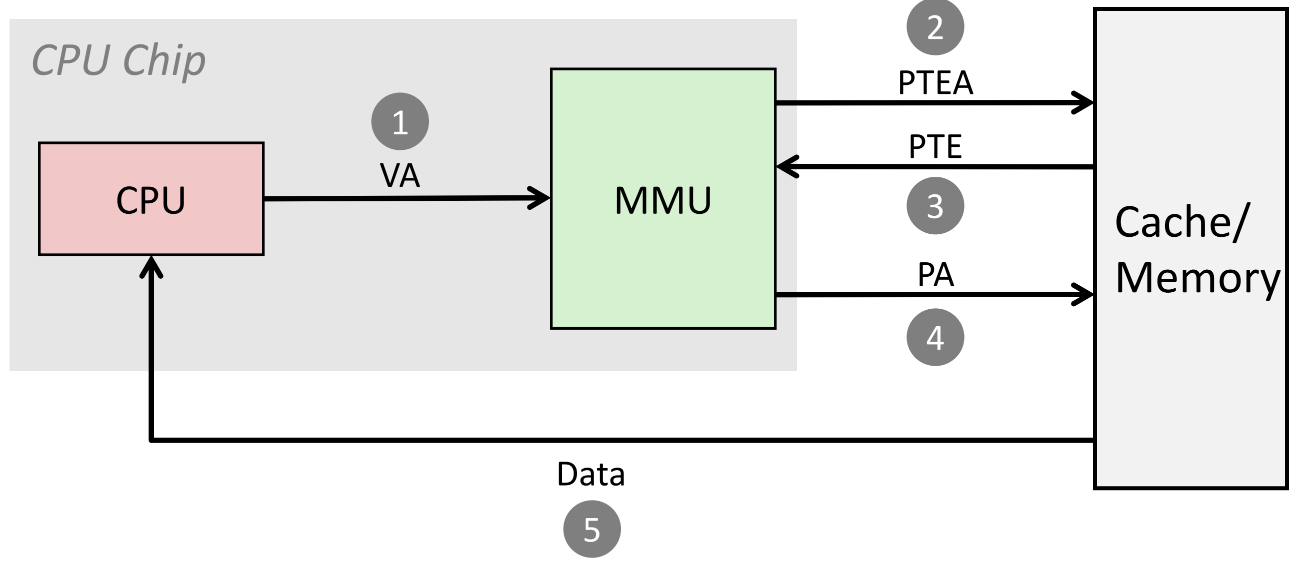
- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address
- Loading
  - **`execve()`** allocates virtual pages for .text and .data sections = creates PTEs marked as invalid
  - The **`.text`** and **`.data`** sections are copied, page by page, on demand by the virtual memory system

VM as a Tool for Mem Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)



Address Translation: Page Hit



1) Processor sends virtual address to MMU

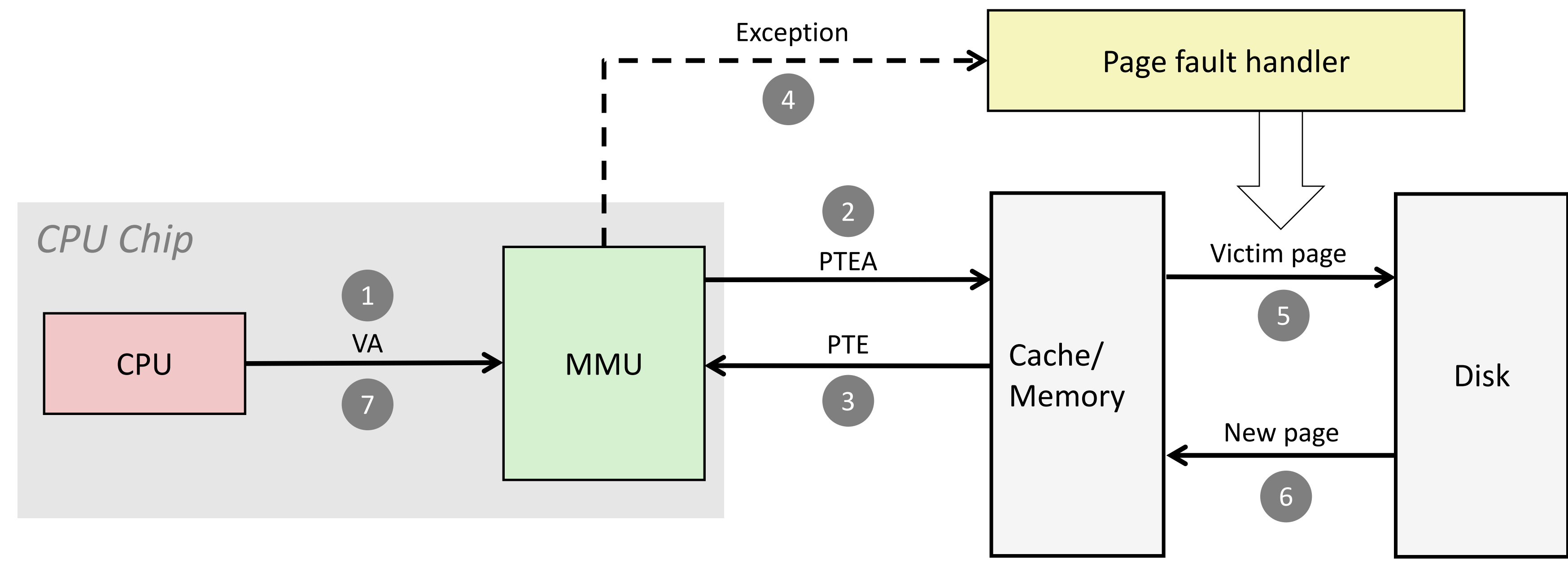
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Address Translation: Page Fault

UCR



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

Integrating VM and Cache

PTE

CPU Chip

PTEA
hit

PTE

PTEA
miss

PTEA

PTEA

CPU

VA

MMU

Memory

PA

PA
miss

PA

Data

PA
hit

L1
cache

Data

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

Elephant(s) in the room

- Problem 1: Translation is slow!
  - Many memory accesses for each memory access
  - Caches are useless!

- **Problem 2: Page table can be gigantic!**
- **We need one for each process**
- **All your memory belong to us!**

"Unfortunately, there's another elephant in the room."

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 requires a small L1 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

TLB Hit

CPU Chip
TLB
2
VPN
PTE
3
1
VA
CPU
MMU
PA
4
Cache/
Memory
Data
5

A TLB hit eliminates a memory access

TLB Miss

CPU Chip

TLB

2 VPN

4 PTE

1 VA

CPU

MMU

3 PTEA

PA 5

Cache/ Memory

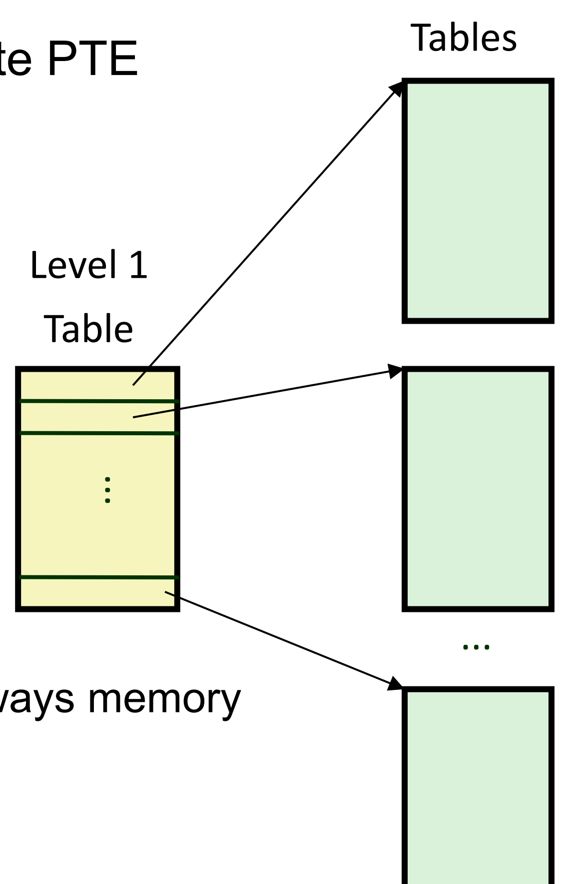
Data 6

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

Fortunately, TLB misses are rare. Why?

Multi-Level Page Tables

- Suppose:
  - 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
- 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

Level 1 page table

- 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

Level 2 page tables

- PTE 0
- ...
- PTE 1023

- PTE 0
- ...
- PTE 1023

- 1023 null PTEs
- PTE 1023

Virtual memory

0

- VP 0
- ...
- VP 1023
- VP 1024
- ...
- VP 2047
- Gap
- 1023 unallocated pages
- VP 9215

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

Advanced Operating Systems (CS 202)

Extensible Operating Systems

Extensibility

- What do we mean by extensibility?
  - Flexible to add new features/functionalities
  - Good efficiency
  - Good security
- Can you give a few examples?
  - Device drivers
  - Browser plugins/extensions

Existing Approaches

- Directly insert code modules
  - E.g., Loadable kernel module
  - Good efficiency
  - Bad security
- Put into a new process
  - E.g., User-mode driver (e.g., FUSE)
  - E.g., Microsoft puts browser plugin into a new process
  - Good security
  - Bad efficiency (context switch/mode switch)

How expensive are border crossings?

- Procedure call: save some general-purpose registers and jump
- Mode switch:
  - Trap or call gate overhead
    - Nowadays syscall/sysreturn
  - Switch to kernel stack
  - Switch some segment registers
- Context switch?
  - Change address space
  - This could be expensive; flush TLB, …

OS design models

- Library OS
- Monolithic Kernel
- Micro Kernel

UCR

OS as library (DOS-like)

Applications

OS Services and Device drivers

Hardware, managed by OS

Monolithic Kernel

Applications

OS Services and Device drivers

What is the difference

Hardware, managed by OS

Micro-kernel

Applications

File System

Memory manage r

CPU schedul er

Micro-kernel

IPC, Address Spaces, …

Hardware, managed by OS

Summary

- DOS-like structure:
  - good performance and extensibility
  - Bad protection
- Monolithic kernels:
  - Good performance and protection
  - Bad extensibility
- Microkernels
  - Very good protection
  - Good extensibility
  - Bad performance!