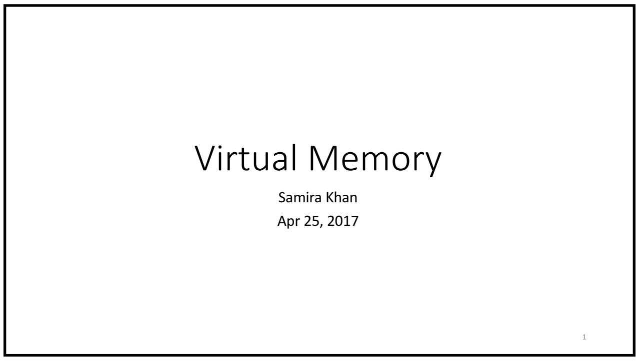
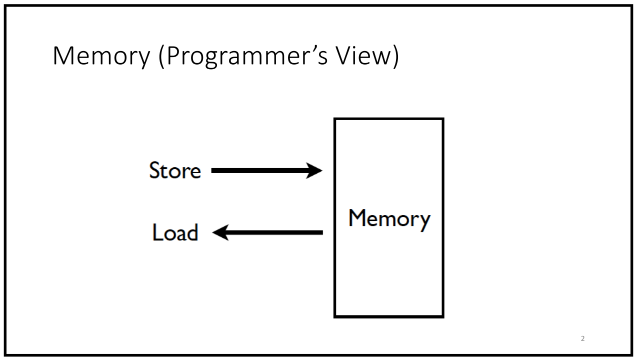


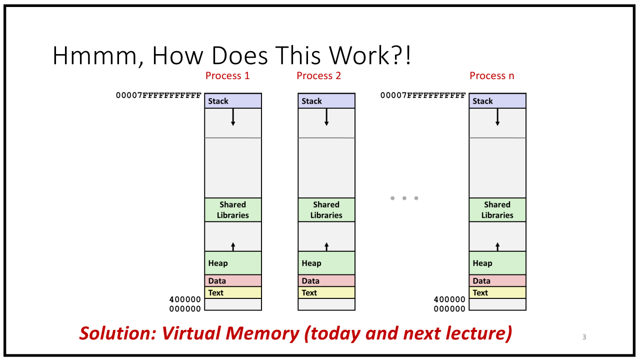
Virtual Memory

Samira Khan

Apr 25, 2017

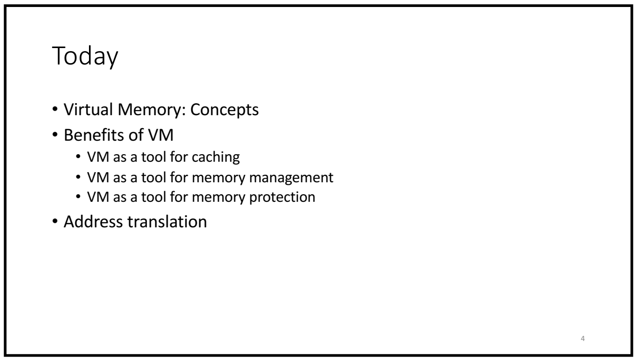


Memory (Programmer's View)



Hmmm, How Does This Work?!

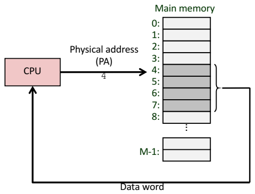
Solution: Virtual Memory (today and next lecture)



Today

- Virtual Memory: Concepts
- Benefits of VM
  - VM as a tool for caching
  - VM as a tool for memory management
  - VM as a tool for memory protection
- Address translation

A System Using Physical Addressing



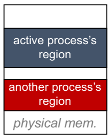
- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

The Problem

- Physical memory is of limited size (cost)
  - What if you need more?
  - Should the programmer be concerned about the size of code/data blocks fitting physical memory?
  - How to manage data movement from disk to physical memory?
  - How to ensure two processes do not use the same physical memory?
- Also, ISA can have an address space greater than the physical memory size
  - E.g., a 64-bit address space with byte addressability
  - What if you do not have enough physical memory?

Difficulties of Direct Physical Addressing

- Programmer needs to manage physical memory space
  - Inconvenient & hard
  - Harder when you have multiple processes
- Difficult to support code and data relocation
- Difficult to support multiple processes
  - Protection and isolation between multiple processes
  - Sharing of physical memory space
- Difficult to support data/code sharing across processes



Virtual Memory

- Idea: Give the programmer the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory
- Programmer can assume he/she has “infinite” amount of physical memory
- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  - Illusion is maintained for each independent process

Abstraction: Virtual vs. Physical Memory

- Programmer sees virtual memory
  - Can assume the memory is "infinite"
- Reality: Physical memory size is much smaller than what the programmer assumes
- The system (system software + hardware, cooperatively) maps virtual memory addresses are to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

+ Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer

-- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

Basic Mechanism

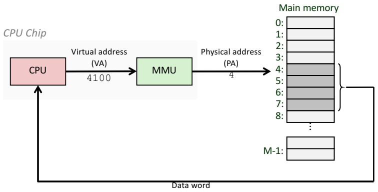
- Indirection (in addressing)

- Address generated by each instruction in a program is a "virtual address"
  - i.e., it is not the physical address used to address main memory

- An "address translation" mechanism maps this address to a "physical address"
  - Address translation mechanism can be implemented in hardware and software together

"At the heart [...] is the notion that 'address' is a concept ***distinct*** *from 'physical location.'" Peter Denning*

A System Using Virtual Addressing



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

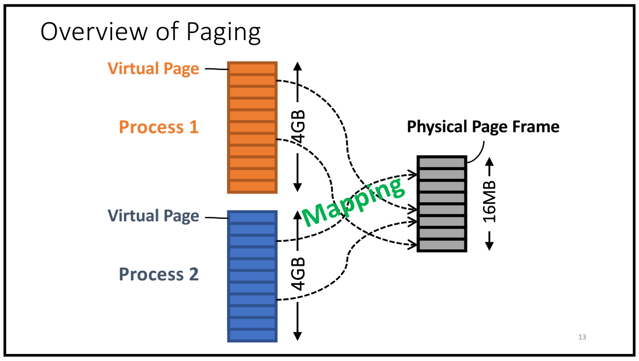
Virtual Pages, Physical Frames

- Virtual address space divided into pages
- Physical address space divided into frames

- A virtual page is mapped to
  - A physical frame, if the page is in physical memory
  - A location in disk, otherwise

- If an accessed virtual page is not in memory, but on disk
  - Virtual memory system brings the page into a physical frame and adjusts the mapping → this is called demand paging

- Page table is the table that stores the mapping of virtual pages to physical frames



Overview of Paging

Virtual Page

Process 1

4GB

Physical Page Frame

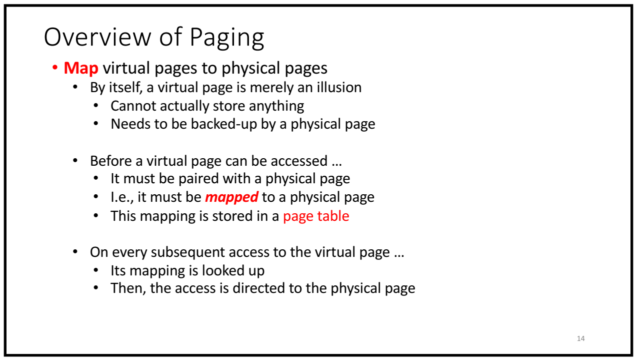
Mapping

16MB

Virtual Page

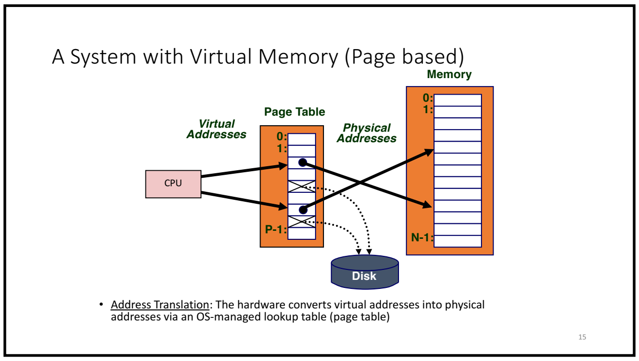
Process 2

4GB



Overview of Paging

- **Map** virtual pages to physical pages
  - By itself, a virtual page is merely an illusion
    - Cannot actually store anything
    - Needs to be backed-up by a physical page
  - Before a virtual page can be accessed …
    - It must be paired with a physical page
    - I.e., it must be ***mapped*** to a physical page
    - This mapping is stored in a page table
  - On every subsequent access to the virtual page …
    - Its mapping is looked up
    - Then, the access is directed to the physical page



A System with Virtual Memory (Page based)

Memory

0:
1:

N-1:

Page Table

0:
1:

P-1:

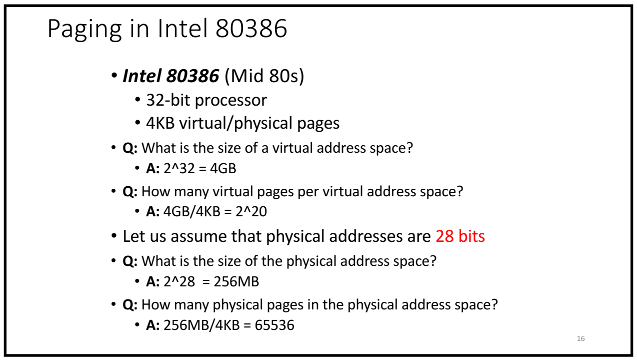
Virtual Addresses

Physical Addresses

CPU

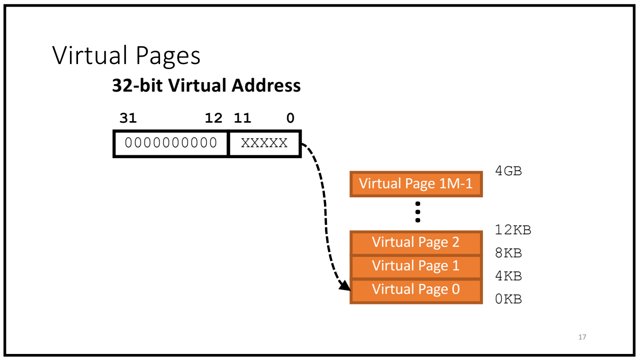
Disk

- Address Translation: The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)



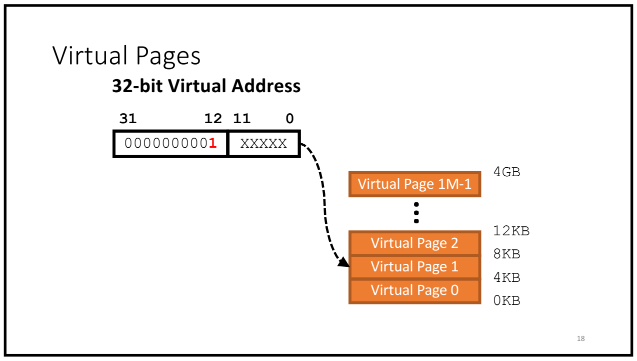
Paging in Intel 80386

- ***Intel 80386*** (Mid 80s)
  - 32-bit processor
  - 4KB virtual/physical pages
- **Q:** What is the size of a virtual address space?
  - **A:** 2^32 = 4GB
- **Q:** How many virtual pages per virtual address space?
  - **A:** 4GB/4KB = 2^20
- Let us assume that physical addresses are 28 bits
- **Q:** What is the size of the physical address space?
  - **A:** 2^28 = 256MB
- **Q:** How many physical pages in the physical address space?
  - **A:** 256MB/4KB = 65536



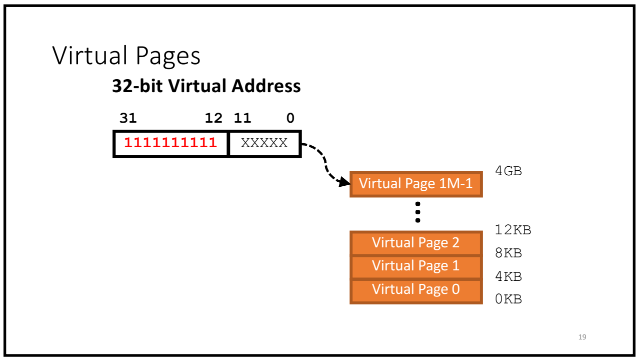
Virtual Pages

32-bit Virtual Address



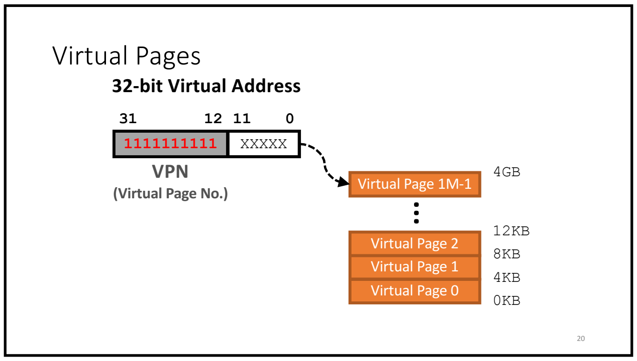
Virtual Pages

32-bit Virtual Address



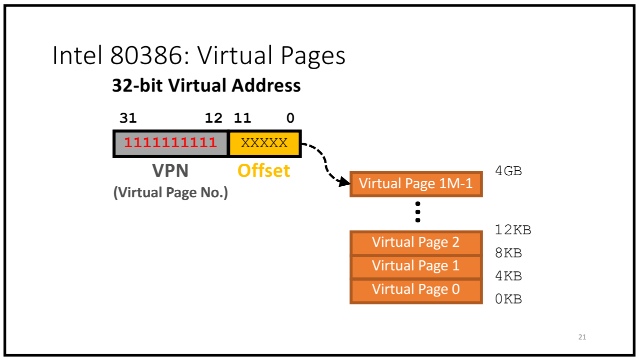
Virtual Pages

32-bit Virtual Address



Virtual Pages

32-bit Virtual Address



Intel 80386: Virtual Pages

32-bit Virtual Address

31 12 11 0

1111111111 XXXXX

VPN **Offset**

(Virtual Page No.)

Virtual Page 1M-1 4GB

⋮

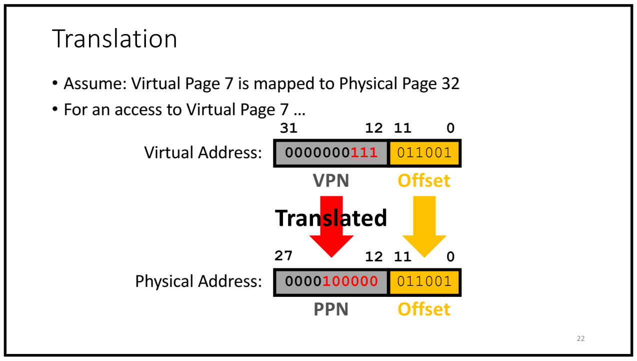
Virtual Page 2 12KB

Virtual Page 1 8KB

Virtual Page 0 4KB

0KB

21



Translation

- Assume: Virtual Page 7 is mapped to Physical Page 32
- For an access to Virtual Page 7 …

31 12 11 0

Virtual Address: 0000000111 011001

VPN **Offset**

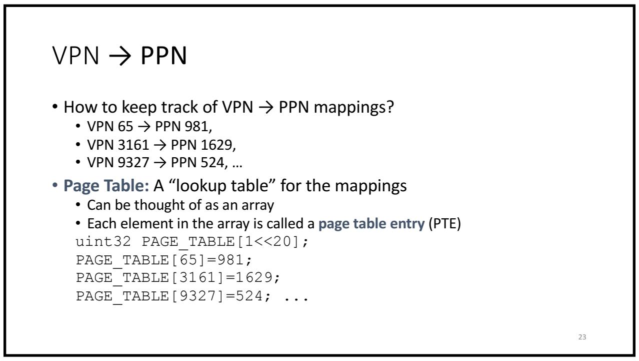
Translated

27 12 11 0

Physical Address: 0000100000 011001

PPN **Offset**

22

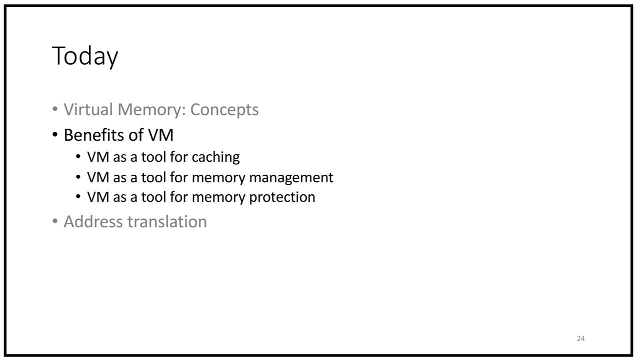


VPN → PPN

- How to keep track of VPN → PPN mappings?
  - VPN 65 → PPN 981,
  - VPN 3161 → PPN 1629,
  - VPN 9327 → PPN 524, …
- **Page Table:** A “lookup table” for the mappings
  - Can be thought of as an array
  - Each element in the array is called a **page table entry** (PTE)

```
uint32 PAGE_TABLE[1<<20];
PAGE_TABLE[65]=981;
PAGE_TABLE[3161]=1629;
PAGE_TABLE[9327]=524; ...
```

23



Today

- Virtual Memory: Concepts
- Benefits of VM
  - VM as a tool for caching
  - VM as a tool for memory management
  - VM as a tool for memory protection
- Address translation

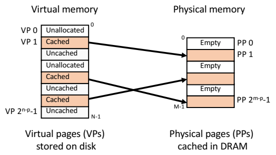
24

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 Caching

- Conceptually, *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)

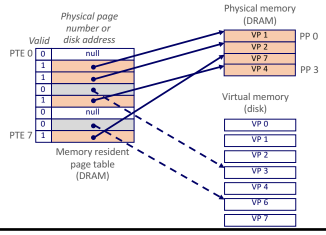


Organization

- 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 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

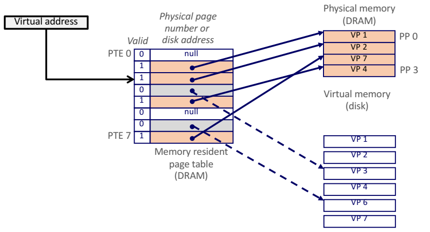
Enabling Data Structure: Page Table

- 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



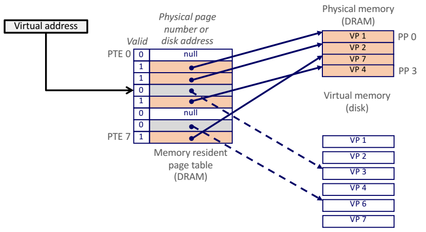
Page Hit

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



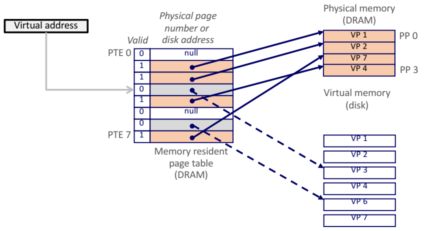
Page Fault

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



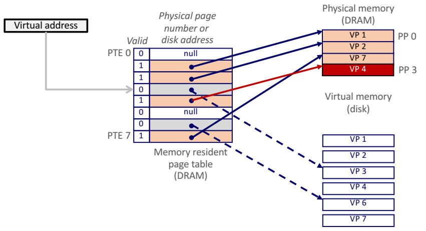
Handling Page Fault

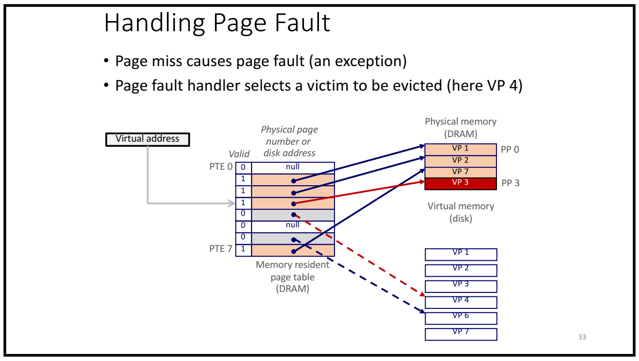
- Page miss causes page fault (an exception)



Handling Page Fault

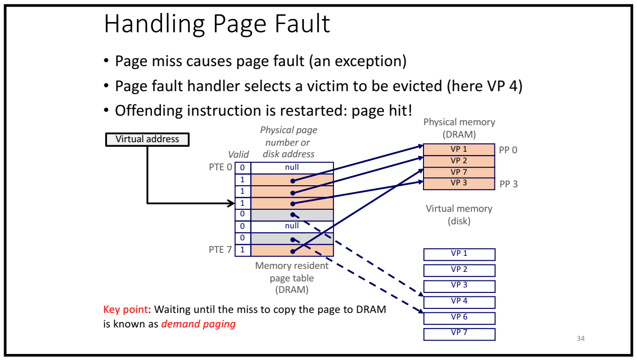
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)





Handling Page Fault

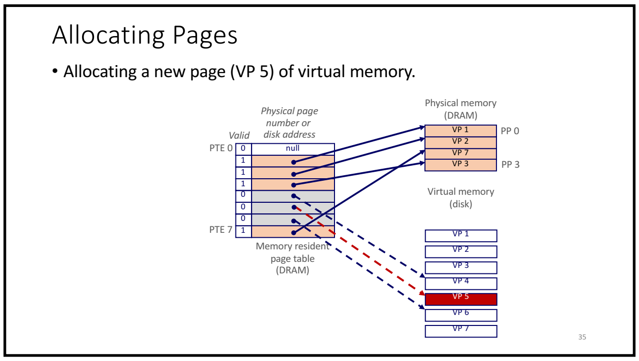
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

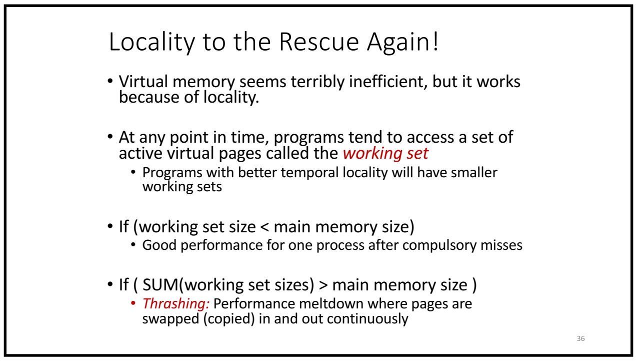
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!

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



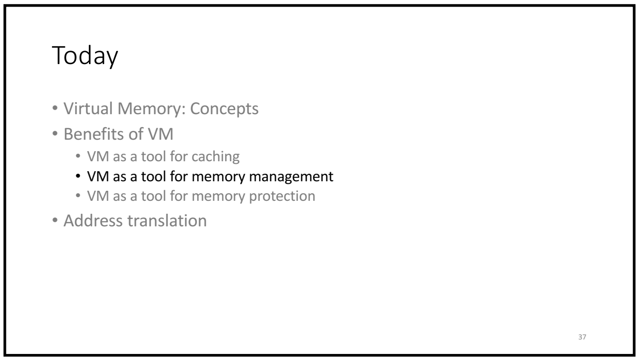
Allocating Pages

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



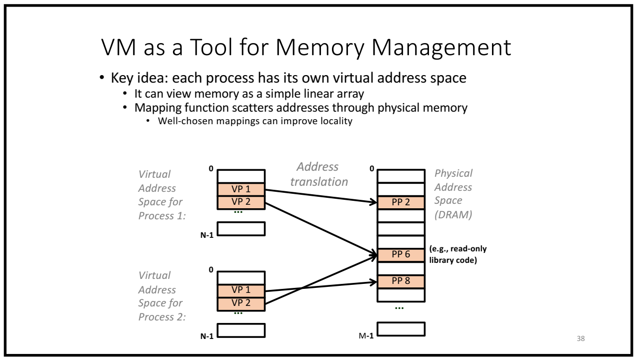
Locality to the Rescue Again!

- 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



Today

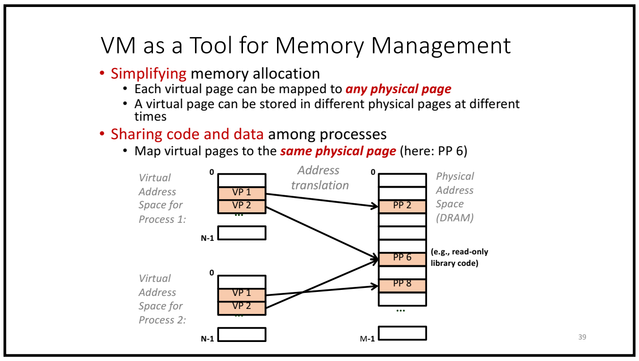
- Virtual Memory: Concepts
- Benefits of VM
  - VM as a tool for caching
  - VM as a tool for memory management
  - VM as a tool for memory protection
- Address translation



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

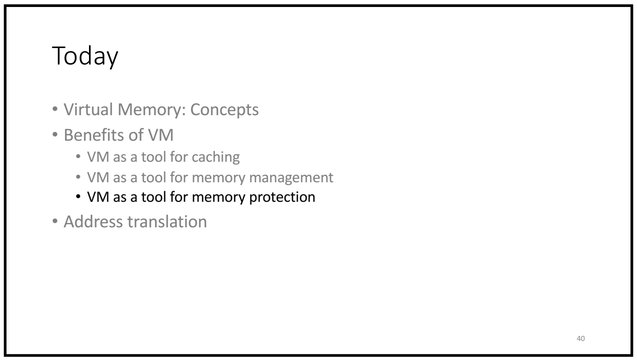
Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
Physical Address Space (DRAM)
PP 6
(e.g., read-only library code)
PP 8
...
M-1
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1



VM as a Tool for Memory Management

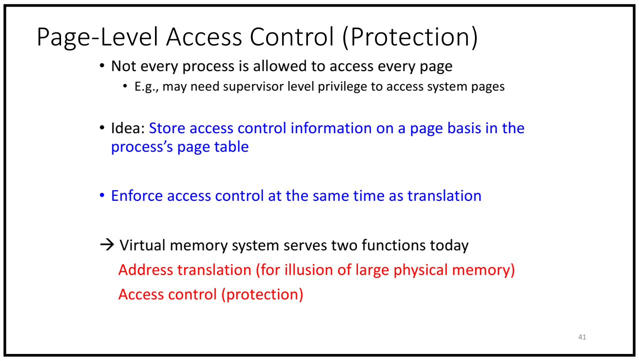
- Simplifying memory allocation
  - Each virtual page can be mapped to ***any physical page***
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the ***same physical page*** (here: PP 6)

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



Today

- Virtual Memory: Concepts
- Benefits of VM
  - VM as a tool for caching
  - VM as a tool for memory management
  - VM as a tool for memory protection
- Address translation



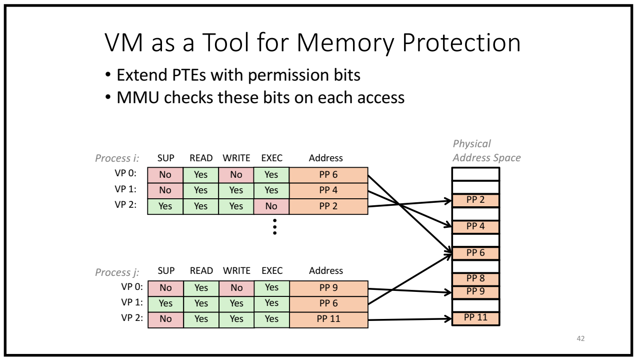
Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages
- Idea: Store access control information on a page basis in the process's page table
- Enforce access control at the same time as translation

→ Virtual memory system serves two functions today

Address translation (for illusion of large physical memory)

Access control (protection)



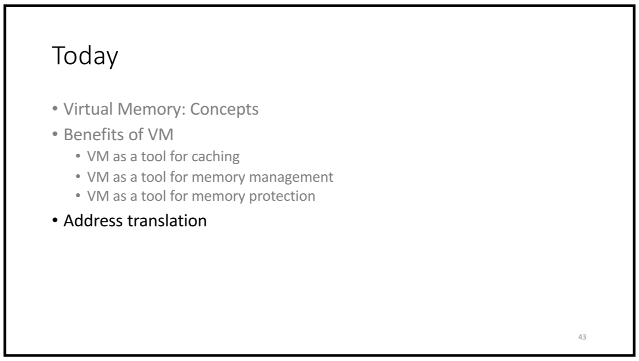
VM as a Tool for Memory Protection

- 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

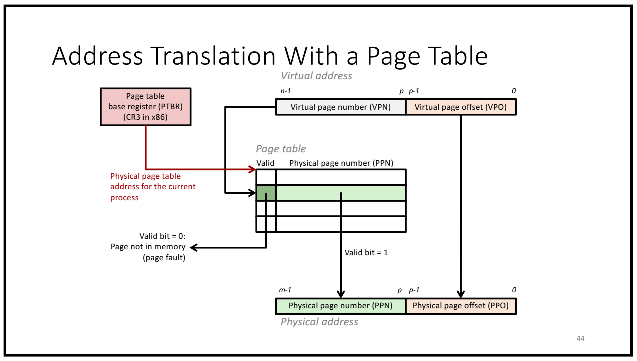
⋮

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



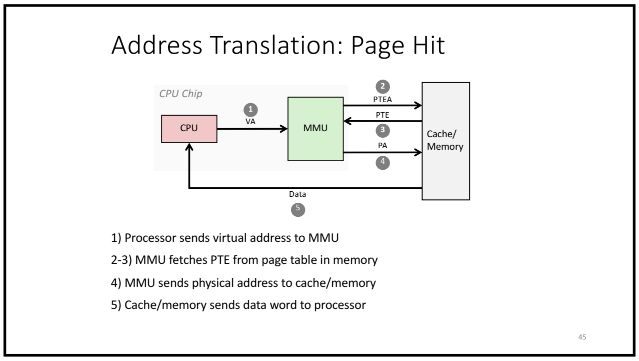
Today

- Virtual Memory: Concepts
- Benefits of VM
  - VM as a tool for caching
  - VM as a tool for memory management
  - VM as a tool for memory protection
- Address translation



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) (CR3 in x86)
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



Address Translation: Page Hit

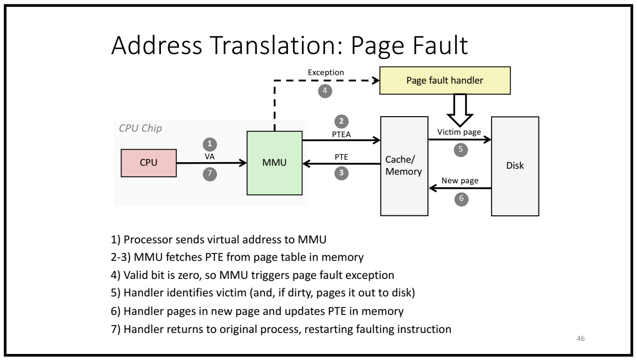
1) Processor sends virtual address to MMU

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

45



Address Translation: Page Fault

1) Processor sends virtual address to MMU

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

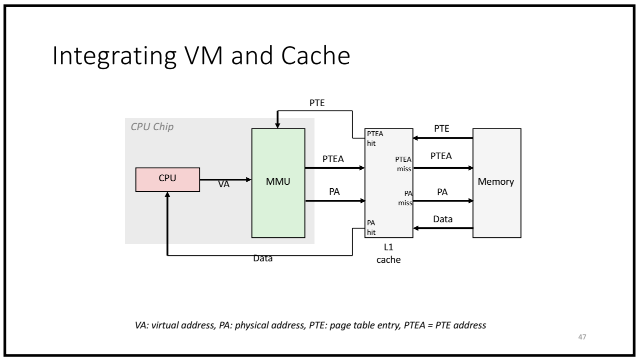
4) Valid bit is zero, so MMU triggers page fault exception

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

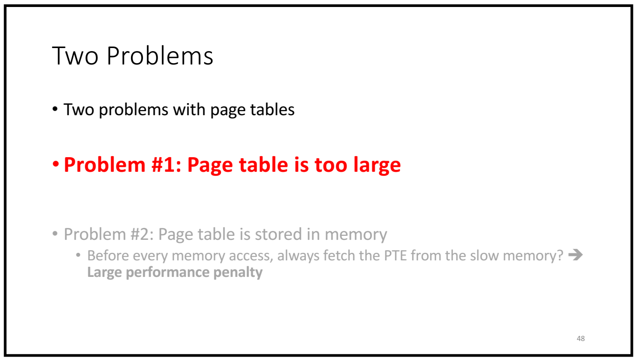
46



Integrating VM and Cache

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

47



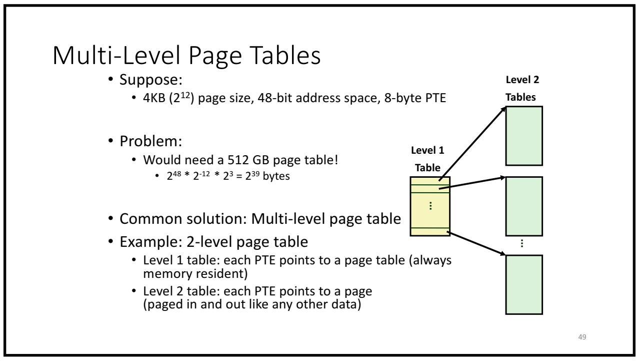
Two Problems

- Two problems with page tables

- **Problem #1: Page table is too large**

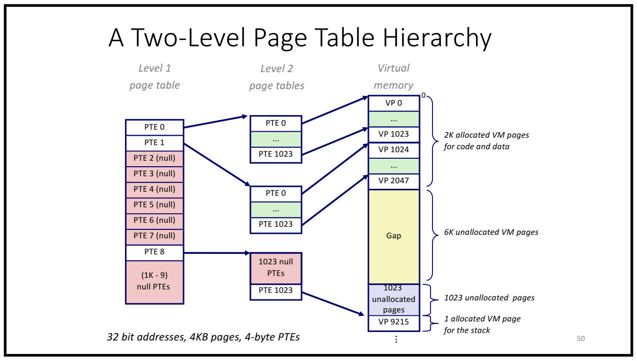
- Problem #2: Page table is stored in memory
  - Before every memory access, always fetch the PTE from the slow memory? ➔ **Large performance penalty**

48



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 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)



A Two-Level Page Table Hierarchy

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