

# CSE 120

# Operating Systems Principles

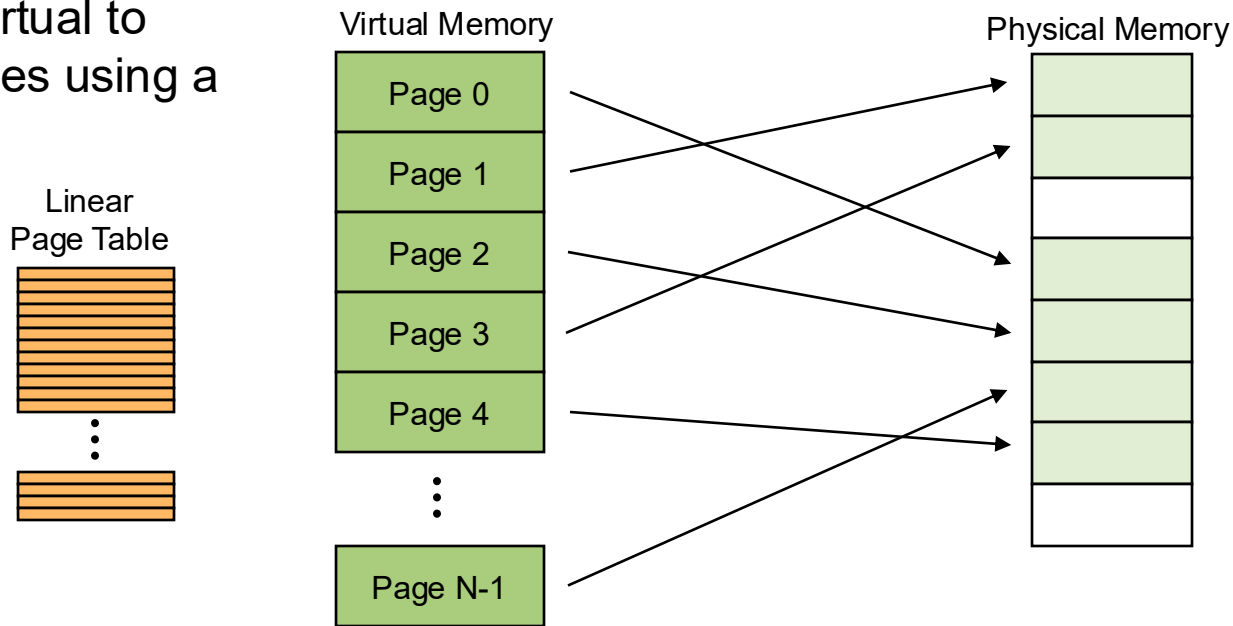
Spring 2025

Lecture 12: TLBs and Swapping

Amy Ousterhout

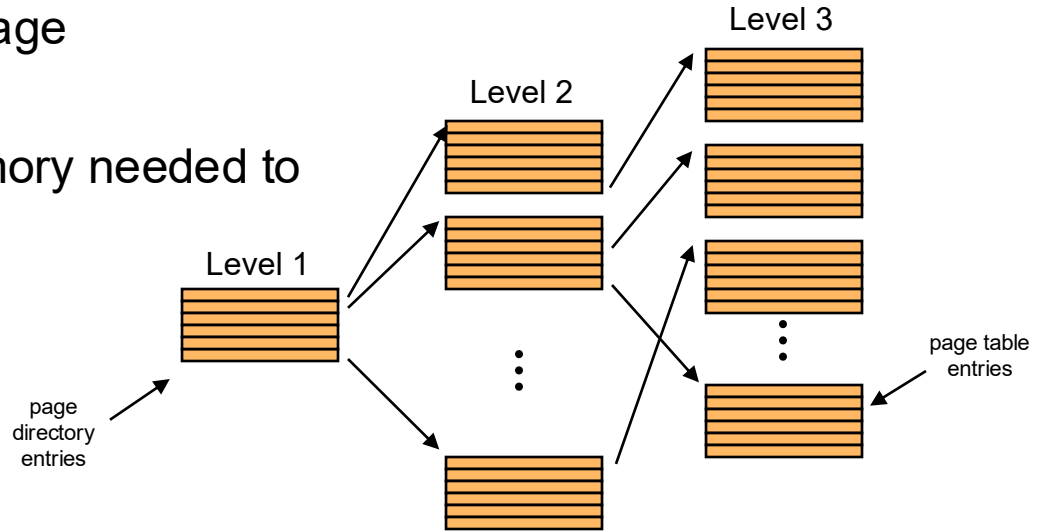
# Paging with Single-Level Page Tables

- Divide physical and virtual memory into fixed-sized chunks called **pages**
- Translate from virtual to physical addresses using a page table



# Multi-Level Page Tables

- Represent page tables hierarchically
- Each page map fits in one page
- Omit empty page maps
- Reduces the amount of memory needed to store page tables



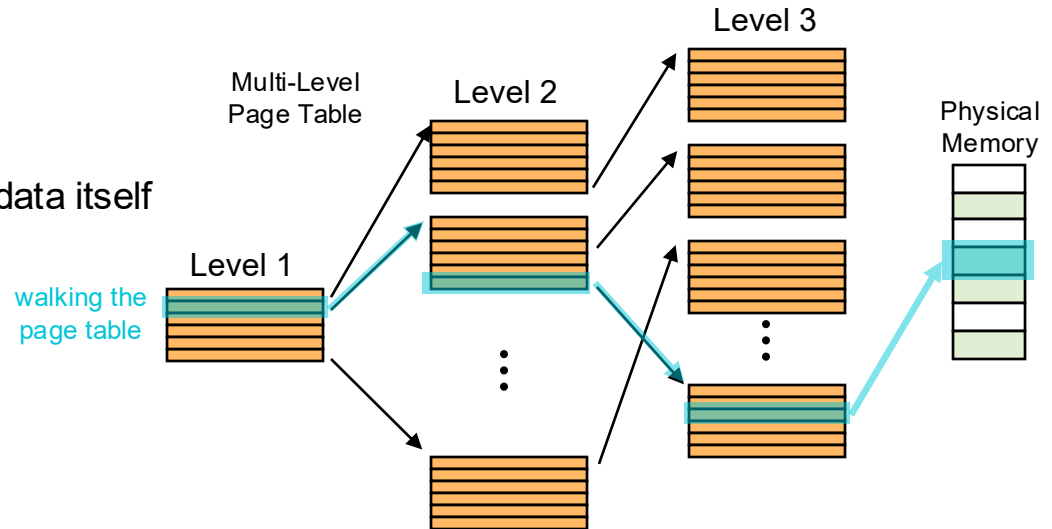
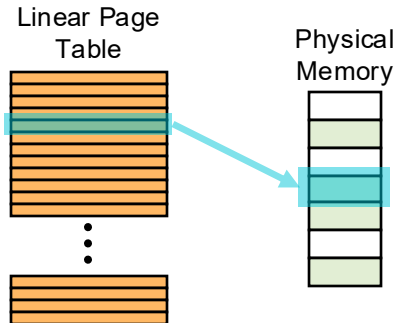
# Today's Outline

---

- Efficient translations
  - Translation Lookaside Buffers (TLBs)
- The illusion of more memory than is physically present
  - Demand-paged virtual memory
- Advanced functionality
  - Shared memory
  - Copy on write
  - Mapped files

# Inefficient Translations

- Programs must translate every virtual address to a physical address
  - On every load or store operation
- How many memory accesses are required per program memory access?
  - Linear page table
    - » 2: page table + data itself
  - N-level page table
    - »  $N + 1$ :  $N$  levels of page maps + data itself



# Translation Lookaside Buffer (TLB)

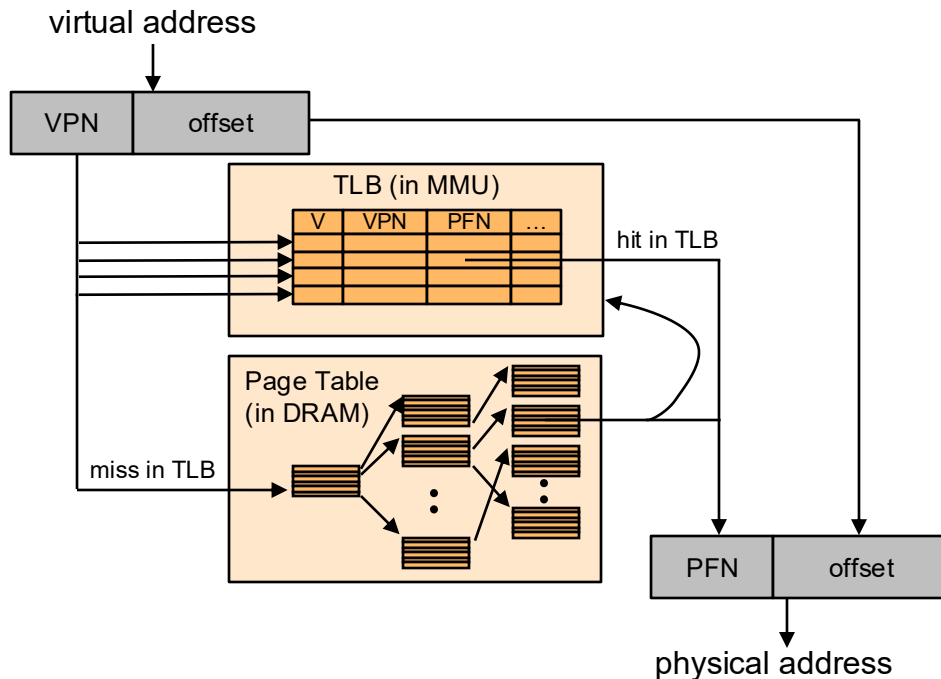
- Solution: take advantage of **locality**
  - In a short period of time, programs tend to access the same page(s) repeatedly
- **Translation Lookaside Buffer**
  - A small hardware cache of recently used translations
  - Each cache entry stores a virtual page number and the corresponding PTE
- Implementation
  - In hardware in the MMU
  - Fully associative cache
  - Typically 64-2048 entries (small!)
  - TLB hits are very fast (~1 CPU cycle)

Valid?	Virtual Page Number	Physical Page Number	...

other fields of  
the PTE

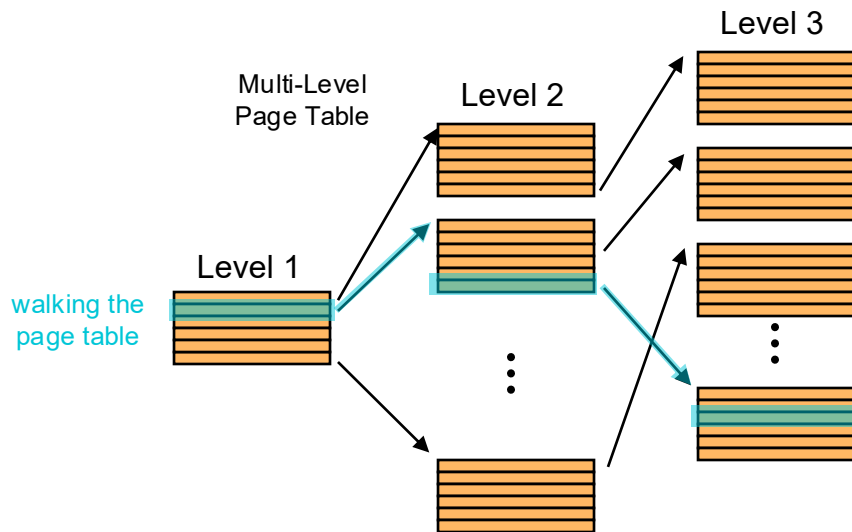
# Address Translation with a TLB

- Compare VPN with every VPN in the TLB
  - Execute all comparisons in parallel
- If there's a hit, use the corresponding PFN
- If there's a miss, consult the page table
  - Look up the PTE to get the PFN
  - Save the PTE in the TLB



# Managing TLBs

- Most address translations are **TLB hits**, handled by the TLB (> 99%)
- **TLB misses** do still occur
  - Need to walk the page table and update the TLB



# Managing TLBs – Handling Misses

---

- What component handles TLB misses?
  - **Hardware** (MMU) [x86, ARM, RISC-V]
    - » Knows where page tables are in main memory
    - » Hardware parses page tables and loads PTE into TLB
    - » Page tables have to match a hardware-defined format (inflexible)
  - **Software** (OS) [MIPS, Alpha, Sparc, PowerPC]
    - » TLB faults to the OS, OS finds the PTE and loads it into the TLB
    - » CPU ISA has instructions for manipulating the TLB
    - » Tables can be in any format (flexible)
- Policy for replacing TLB entries
  - Random, Least Recently Used

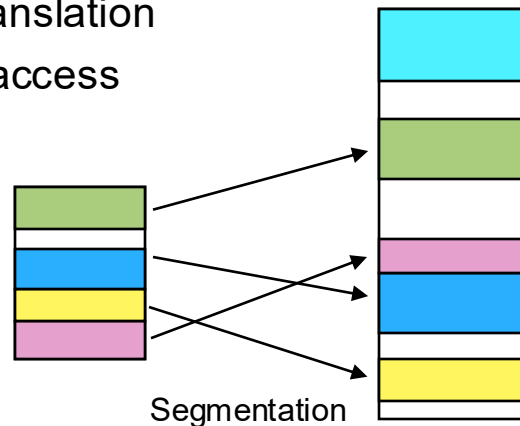
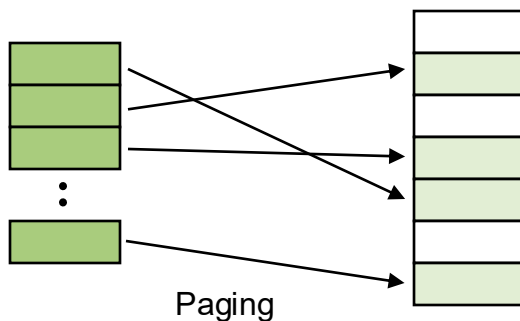
# Managing TLBs

---

- Handling **context switches**
  - Invalidate all TLB entries
    - » Lots of TLB misses afterward
  - Tag each entry with an Address Space Identifier (ASID)
    - » Check each TLB entry against a register containing the process id of the currently executing process
    - » Update this register on a context switch
- Maintaining **consistency** between the TLBs and page tables
  - When the OS modifies a PTE (e.g., changes protection bits) it needs to invalidate the PTE if it is in the TLB
  - On multi-core CPUs, must invalidate PTE entries on all cores (**TLB shoot-down**)

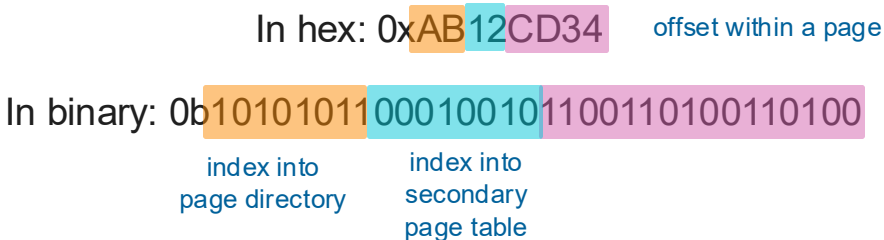
# Poll – Segmentation vs. Paging

- Which of the following is *not* a correct statement about segmentation and paging?
  - A: there is less external fragmentation with paging than with segmentation
  - B: with both approaches, an array allocated on the stack might not be contiguous in physical memory (with segmentation, assume one segment for the stack)
  - C: there is typically less internal fragmentation with paging than with segmentation
  - D: both approaches compute an offset as part of address translation
  - E: both approaches enforce permissions on every memory access



# Poll – Multi-Level Page Tables

- Consider a 32-bit virtual address space with 64 KB pages ( $2^{16}$  bytes).
- Assume two-level page tables where directory and secondary page tables have the same number of entries.
- For the virtual address 0xAB12CD34, what is the index of the page table entry in the secondary page table?
  - A: 0xCD34
  - B: 0xAB
  - C: 0xAB12
  - D: 0x12
  - E: 0x120000



# Today's Outline

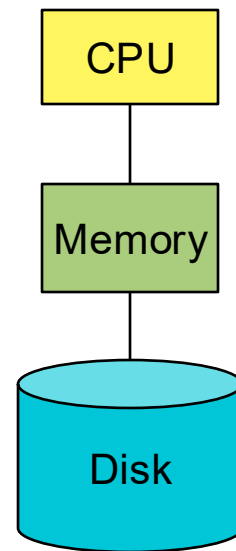
---

- Efficient translations
  - Translation Lookaside Buffers (TLBs)
- The illusion of more memory than is physically present
  - Demand-paged virtual memory
- Advanced functionality
  - Shared memory
  - Copy on write
  - Mapped files

# Limited Space in Physical Memory

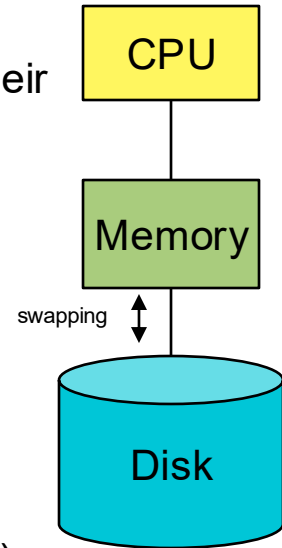
---

- What if all of our data doesn't fit in memory at once?
- DRAM (memory)
  - Low latency
  - High bandwidth
  - Limited capacity (tens of GB)
- Disk
  - Much larger capacity (a few TB)
  - 100,000x higher latency
  - 1,000x less bandwidth
- Goal: use disk to make physical memory appear larger than it is

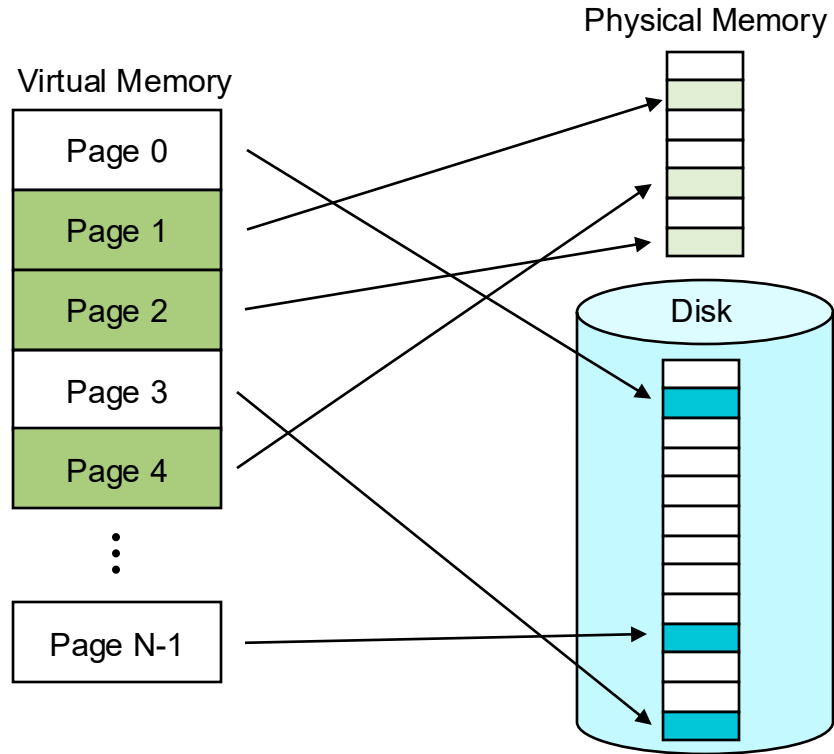


# Demand Paging

- Store data on disk and **use main memory as a cache**
  - Take advantage of **locality** – programs typically spend most of their time using a small fraction of their data
  - Keep data in physical memory while it is used
  - Keep unused data on disk in a **swap file**
    - » Also called a backing store, swap space, etc.
- **Demand paging**: load pages on demand
  - Each virtual page is either in memory or on disk
  - Allocate page in memory when first accessed
  - **Swap** pages in from disk when they are accessed (if not present)
- Ideal: memory system behaves as though it has the performance of main memory but the cost and capacity of disk

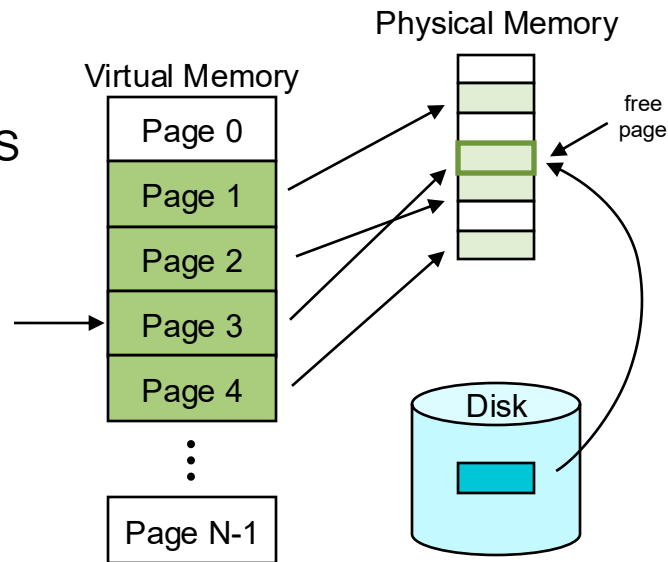


# Paged Virtual Memory



# Page Faults - Overview

- What happens when a process references a page that is in the swap file?
  - Process references memory (e.g., MOV)
  - Valid/present bit in PTE is set to 0
    - » Indicates that the page is not accessible
  - Triggers an exception (**page fault**) and a trap to the OS
  - OS runs the page fault handler
    - » Finds a free page frame in physical memory
    - » Reads the page in from the swap file to the page frame
    - » OS updates the PTE, sets valid=1
    - » Returns to the process
  - Process re-executes the memory reference



# Page Faults - Details

---

- How does the OS know which page caused the page fault?
  - Hardware saves the faulting address in a special register
- What if there are no available page frames?
  - Evict a page to disk
- How does the OS know where the page is located in the swap file?
  - Store this information in the PTE (it's invalid anyway)
- Where should we resume process execution?
  - Restart the faulting instruction

# Paging Policies

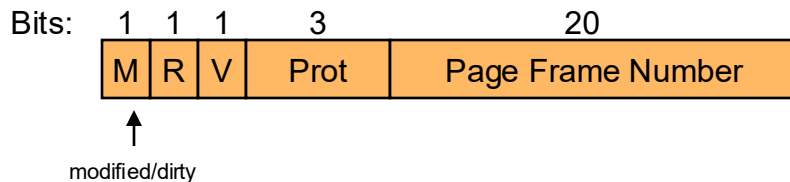
---

- **Page fetching**: when to bring pages into memory
  - **Demand paging** – don't load a page until it is referenced
  - **Prefetching** – try to predict when pages will be needed and load them in advance
    - » Simple approach: during a page fault, bring in the next N pages
- **Page replacement**: which pages to evict from memory
  - Next lecture

# Page Frame Questions

---

- Is it safe for the OS to reclaim a physical page from process A and grant it to process B?
  - Yes, but the OS must **zero the page** first if it's a newly allocated page
- When we evict a page, do we always need to write the contents to the disk?
  - **Only modified (dirty) pages need to be written to disk**
  - Clean pages do not (they're already on disk)
  - Use the modified/dirty bit in the PTE



# Causes of Faults

- PTE can indicate the type of fault:

- **Page fault** – PTE indicates not valid

- » Page not in physical memory (in swap file)
- » Virtual page allocated by program but not yet accessed
- » Virtual page not allocated (invalid memory access)

- **Protection fault** – read/write/execute permissions don't match access type

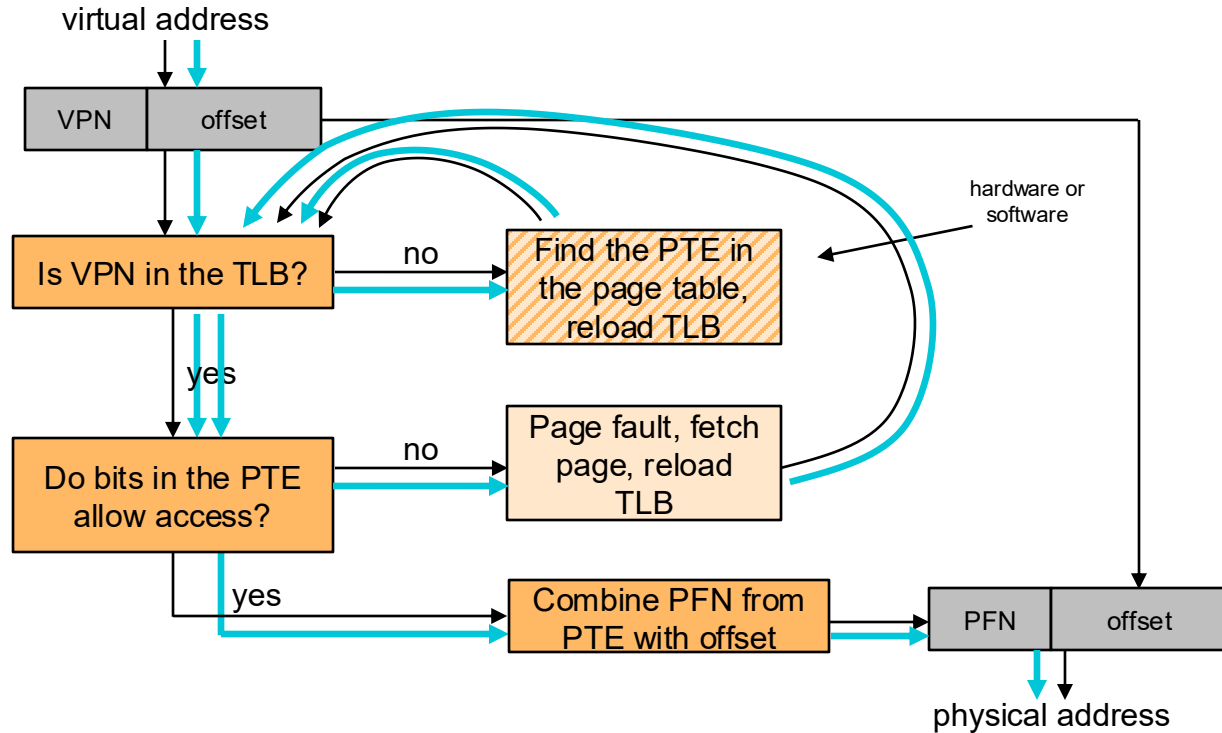
- » Operation not permitted on page

- OS approaches to handling page faults or protection:

- Fetch page from the swap file
- Allocate a physical page
- Send a fault back to process (e.g., **segmentation fault**)
- Other advanced functionality



# Address Translation for a Swapped-Out Page



# Today's Outline

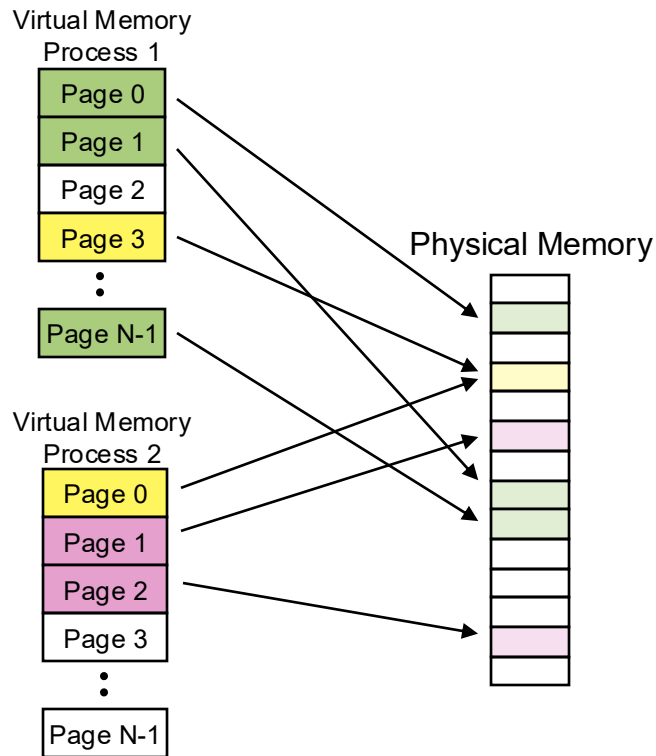
---

- Efficient translations
  - Translation Lookaside Buffers (TLBs)
- The illusion of more memory than is physically present
  - Demand-paged virtual memory
- **Advanced functionality**
  - Shared memory
  - Copy on write
  - Mapped files



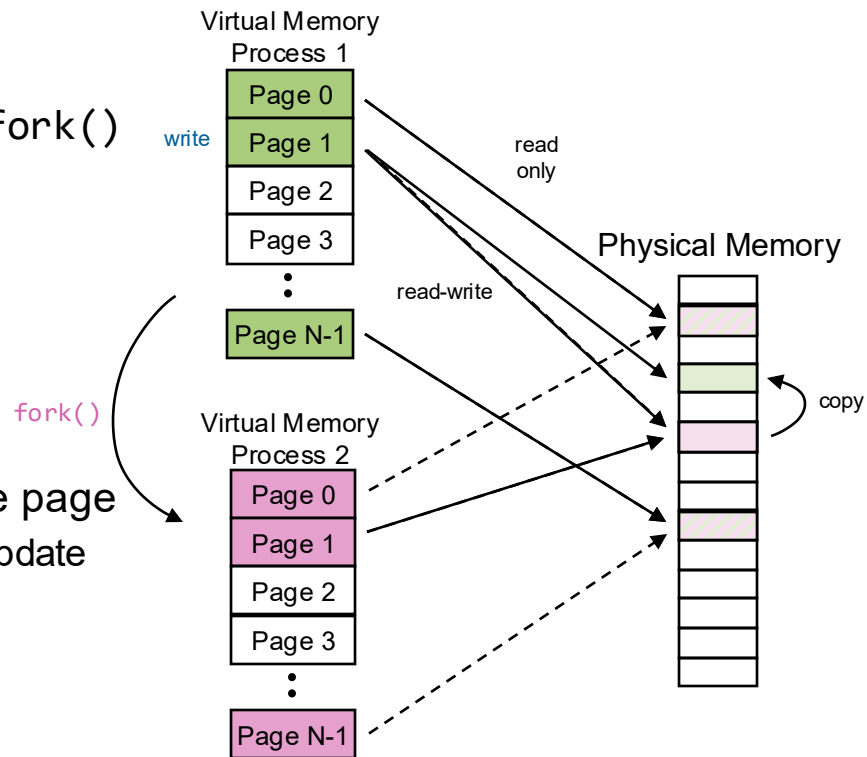
# Shared Memory

- Use **shared memory** to allow multiple processes to access the same memory
- Use a system call to set up shared memory
  - Unix: `shm_open`, `shm_unlink`
- Have PTEs in both page tables map to the same physical frame
- Can map shared memory at the same or at different virtual addresses in each process address space



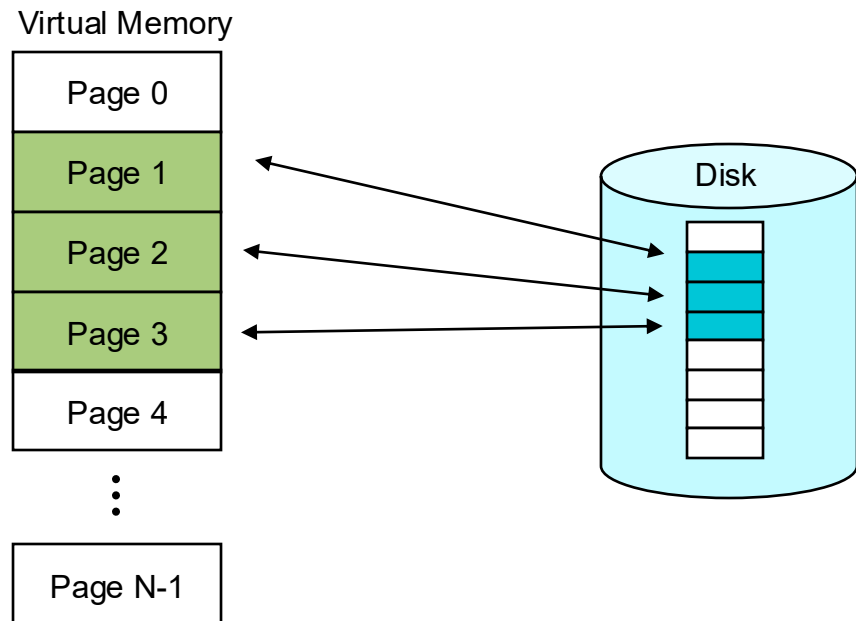
# Copy on Write

- OSES spend a lot of time copying data
  - E.g., entire address spaces to implement `fork()`
- We would like to avoid copying until it's necessary
- **Copy-on-Write**
  - Share pages as read-only, using shared mappings
  - Only copy when a process tries to write the page
    - » Protection fault, trap to the OS, copy page, update page mapping, restart write instruction



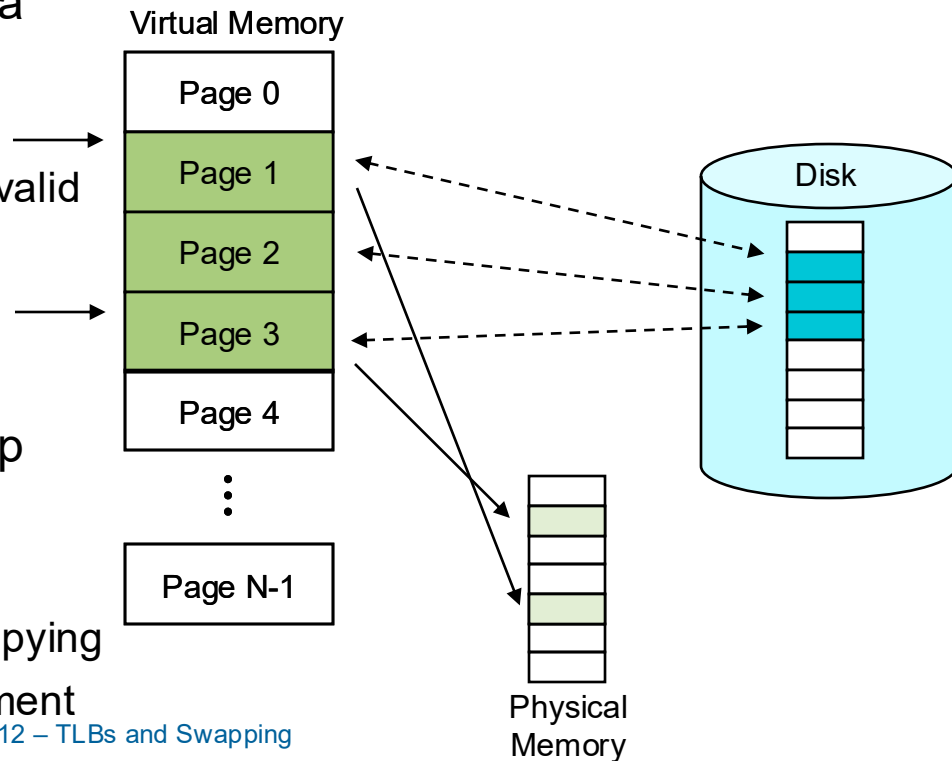
# Mapped Files

- Familiar I/O interface:
  - `open()`
  - `read(fd, buf, count)`
  - `close()`
- With mapped files:
  - Map a file into the virtual address space
  - Process can access file in the same way as memory
    - » Regular load and store instructions (e.g., MOV)



# Mapped Files – Details

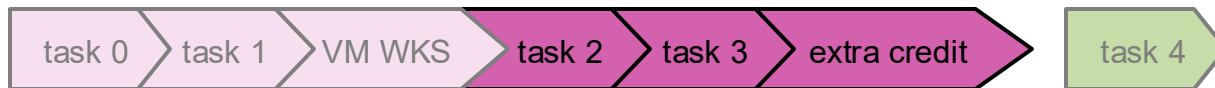
- Process sets up the mapping using a system call
  - Unix: `mmap`
  - OS creates PTEs, but marks them invalid
- On page access:
  - OS brings the page into memory
  - Updates PTE
- OS can evict pages to disk and swap back in as usual
- Tradeoffs:
  - Access files just like memory, less copying
  - Process does not control data movement



# Upcoming Tasks

---

- Read chapters 17 and 22
- Project 2
  - Due Friday 5/16 at 11:59 pm



- Homework 3
  - Due Tuesday 5/20 at 11:59 pm