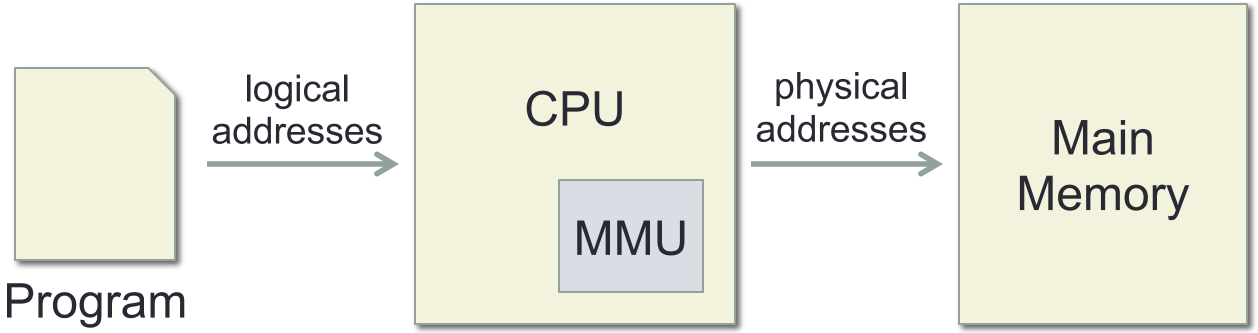
PROCESS VIRTUAL MEMORY PART 2

CS124 – Operating Systems

Spring 2024, Lecture 16

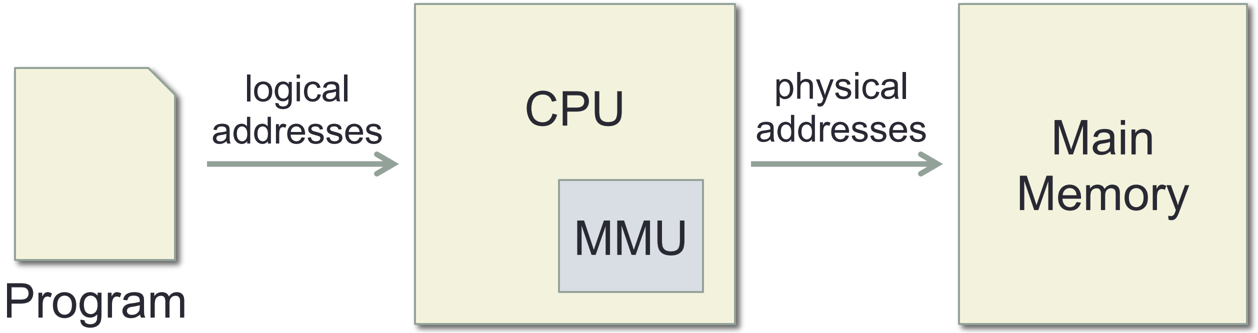
Virtual Memory Abstraction

- Last time, officially introduced concept of virtual memory
- Programs use virtual addresses (a.k.a. logical addresses) to reference instructions, variables, the memory heap, etc.
- The processor translates these into physical addresses using the Memory Management Unit
- Many different ways that the MMU can perform this translation
  - e.g. relocation register, memory segments, paging



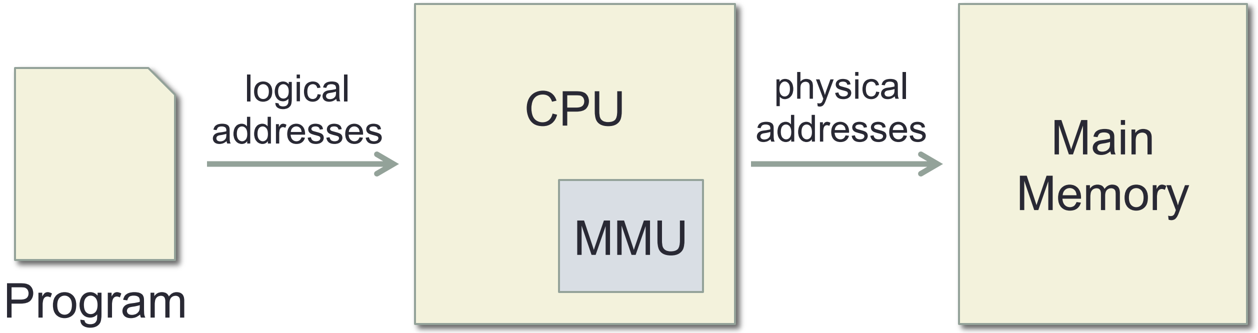
Virtual Memory Abstraction (2)

- Most processors currently use paging to map virtual addresses to physical addresses
  - Virtual and physical memory are divided into uniform blocks
  - Virtual memory pages are mapped to physical page frames
- Again, many ways to manage the mapping of virtual to physical pages
  - Simple page tables, hierarchical page tables, hashed page tables, inverted page tables



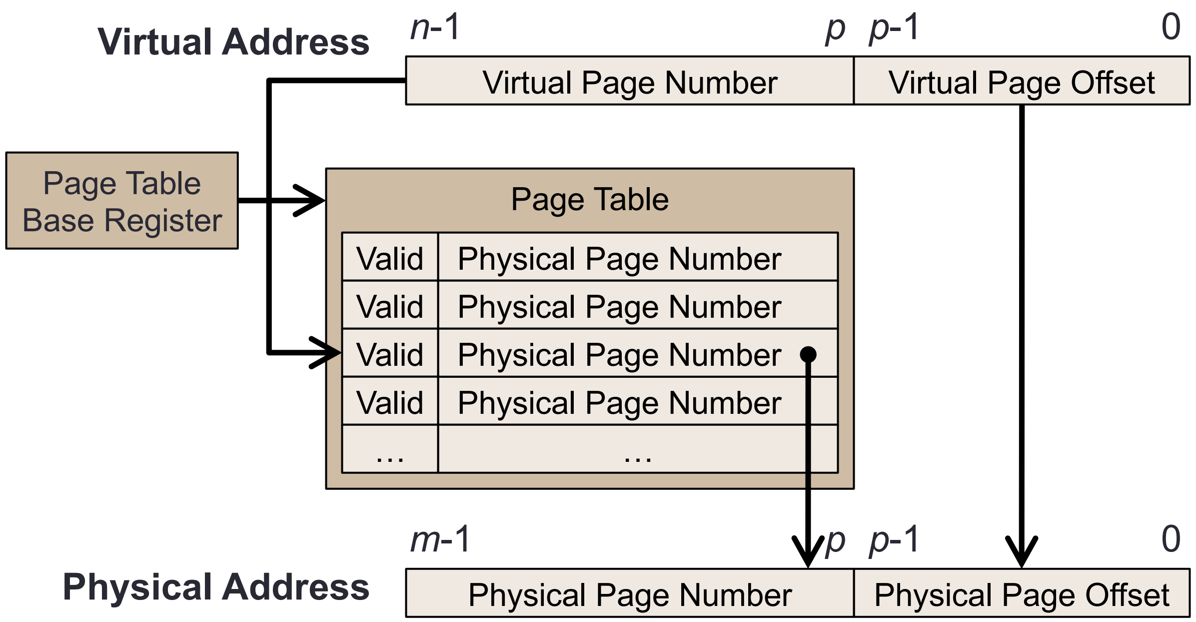
Virtual Memory Abstraction (3)

- Several big benefits from virtual memory abstraction
  - Process isolation is extremely important for multitasking systems
  - Simplified application binary interface (ABI)
- One of the greatest benefits is the ability to move pages to and from a backing store (e.g. SSD or hard disk)
  - Allows programs to use much more memory than the system’s actual physical memory size
  - Observation: programs don’t always access *all of* their memory…
    Move unused pages to a backing store to free up physical memory



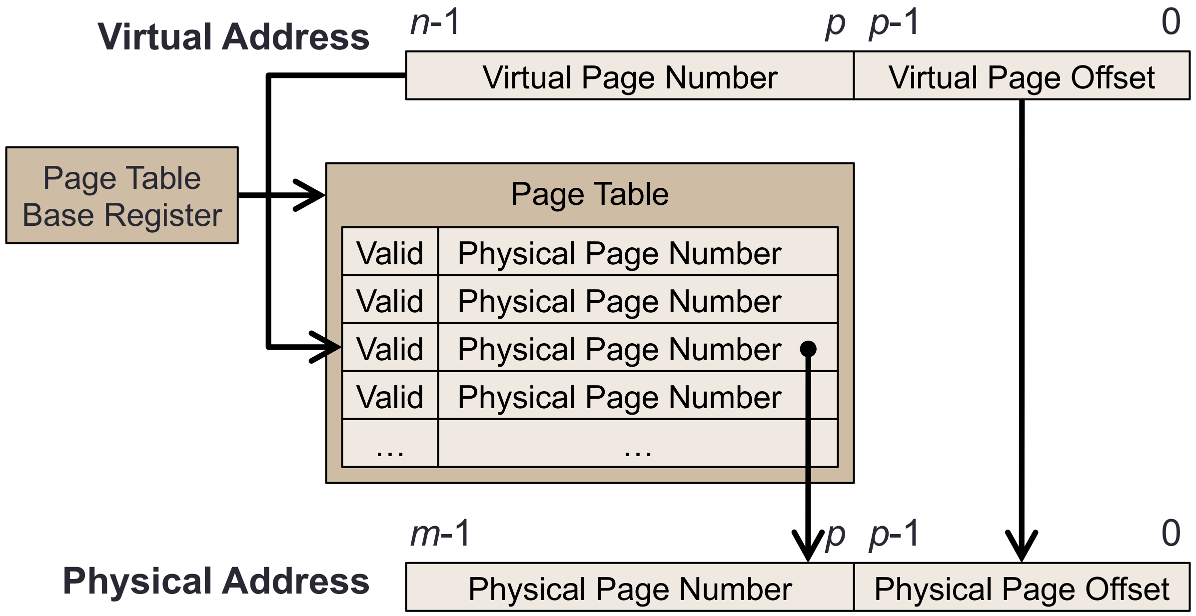
Virtual Memory and Paging

- To support moving pages between physical memory and the backing store, must extend page tables with more info
- Need a **valid/invalid bit** for every entry in the page table
  - "Valid" indicates the virtual page corresponds to a physical frame, and thus is in memory
  - "Invalid" indicates that the page doesn't currently map to a frame in memory
  - IA32 calls this bit "present" (makes more sense)
- If "valid" bit is 0, MMU ignores the rest of page table entry
  - OS can store its own details there if it wishes to



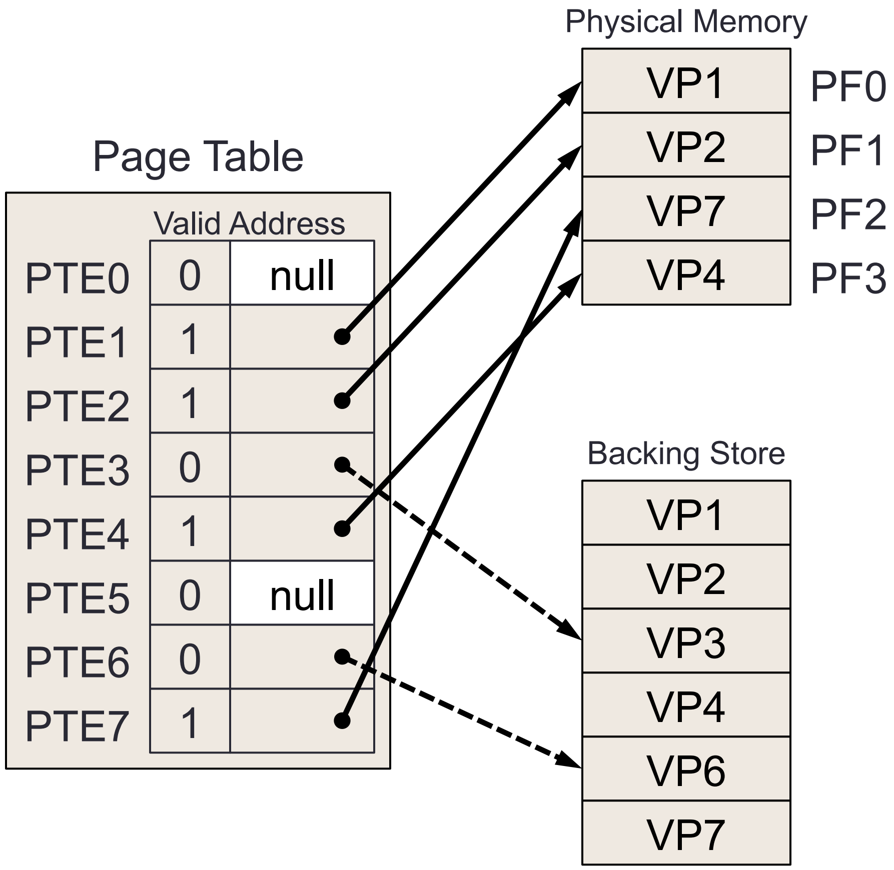
Virtual Memory and Paging (2)

- When MMU translates a memory access, it examines this valid/invalid bit
  - If bit is "valid," MMU can handle address translation all by itself
  - If bit is "invalid," MMU cannot proceed!
- MMU generates a page fault, allowing the OS kernel to resolve the fault (if it can be resolved)



Virtual Memory Example

- Example: small virtual memory
  - 4 physical page frames (PF), 8 virtual pages (VP)
- Some virtual pages are in physical memory
  - VP1, VP2, VP4, VP7
- Some virtual pages are only in the backing store
  - VP3, VP6
- Two virtual pages have not been allocated
  - VP0, VP5



Page Table

	Valid	Address
PTE0	0	null
PTE1	1	
PTE2	1	
PTE3	0	
PTE4	1	
PTE5	0	null
PTE6	0	
PTE7	1	

Physical Memory

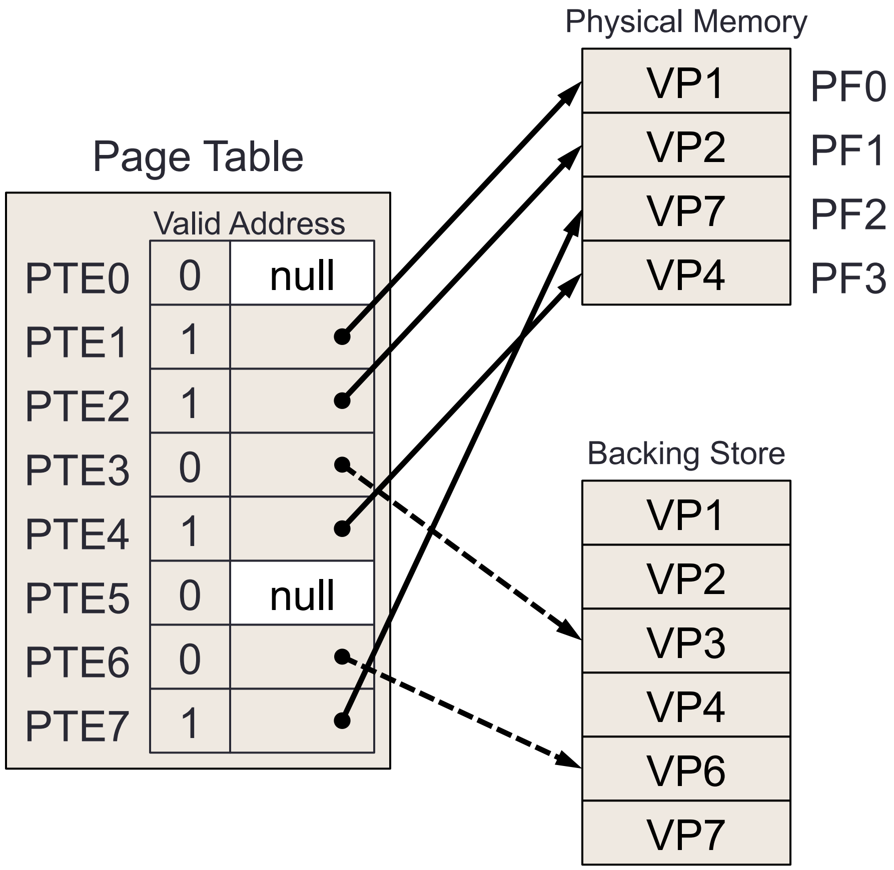
VP1	PF0
VP2	PF1
VP7	PF2
VP4	PF3

Backing Store

VP1
VP2
VP3
VP4
VP6
VP7

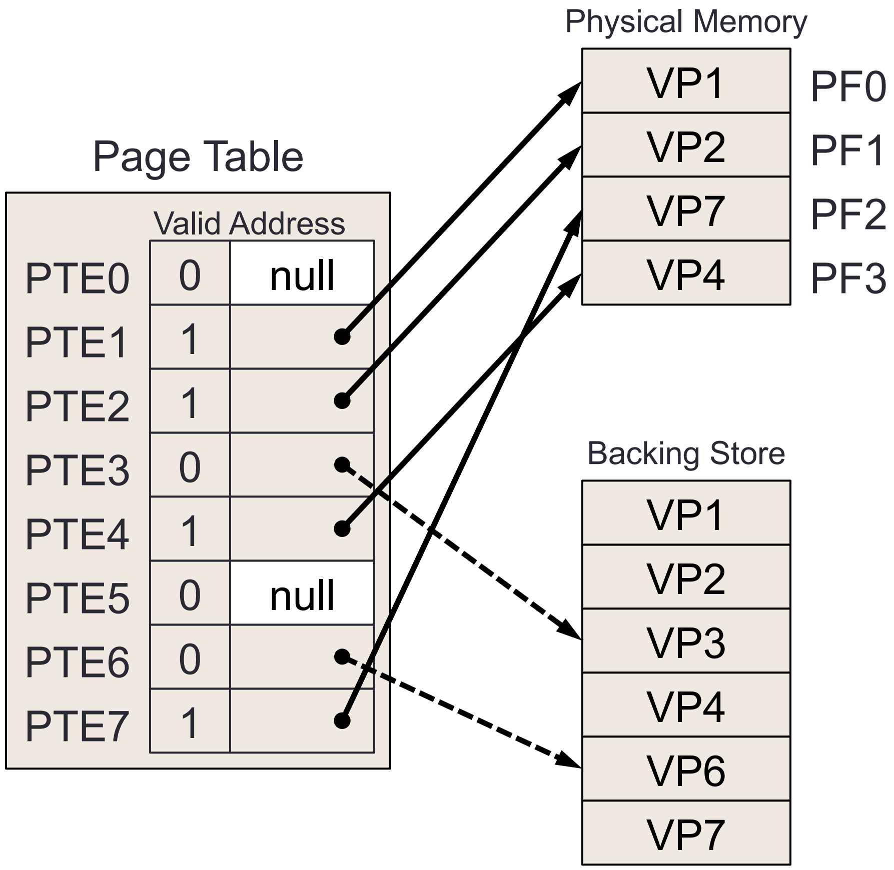
Virtual Memory Accesses

- Program accesses a word in Virtual Page 2
  - MMU looks in page table for virtual page 2 (PTE2)
- “Valid” flag is 1:
  - Page is in physical memory
- Virtual page 2 is stored in physical frame 1…
- MMU uses entry to translate the virtual address
  - Physical frame number is used to generate the physical address
  - Physical address is sent to main memory



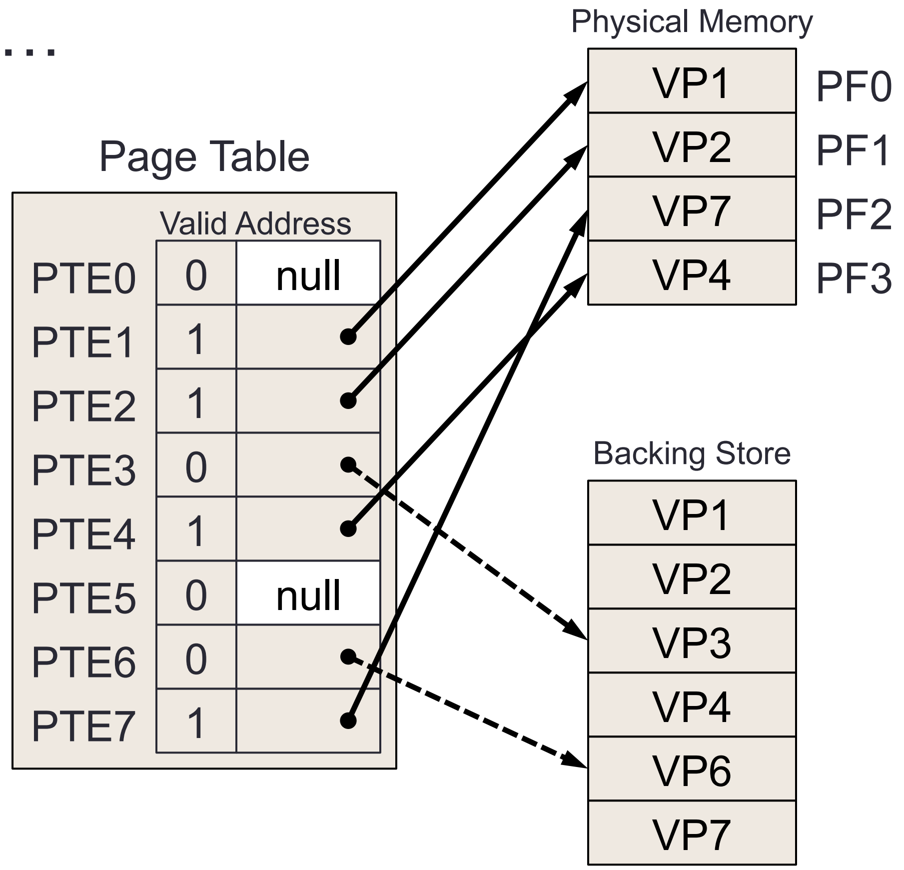
Virtual Memory Accesses (2)

- Next, program accesses a word in Virtual Page 6
  - Again, MMU looks in page table for VP6, but valid flag is 0
  - MMU cannot satisfy the request…
- MMU generates a page fault to allow the kernel to resolve the issue
- Kernel handler sees that VP6 is on the backing store
  - Can move this page back into memory
- Problem: no frame is available to hold the virtual page



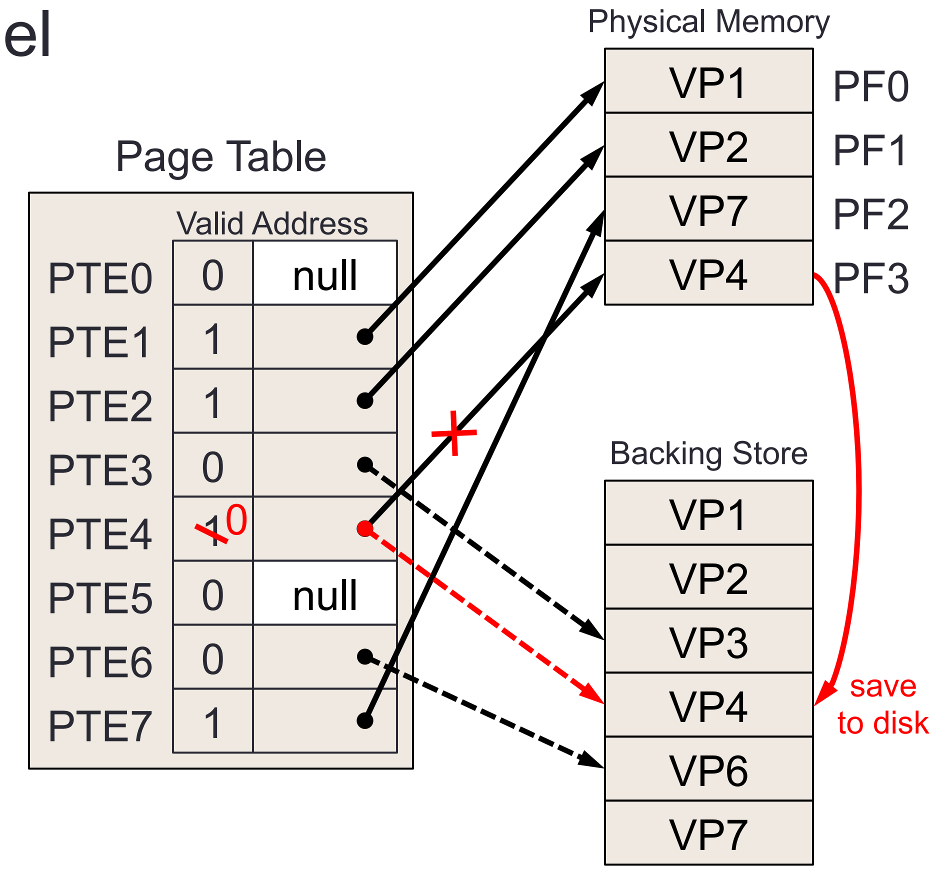
Virtual Memory Accesses (3)

- Kernel must select a victim page to evict from memory
  - e.g. kernel selects VP4 as the victim page
- Want to avoid writing VP4 to disk if it didn't change…
  - Both physical memory and the swap disk have a copy of VP4
  - If two versions of VP4 are the same, why write it back?
  - (Disk accesses are SLOW)
- Extend page table to also include a **dirty bit**
  - MMU sets this bit to 1 when a valid virtual page is written to



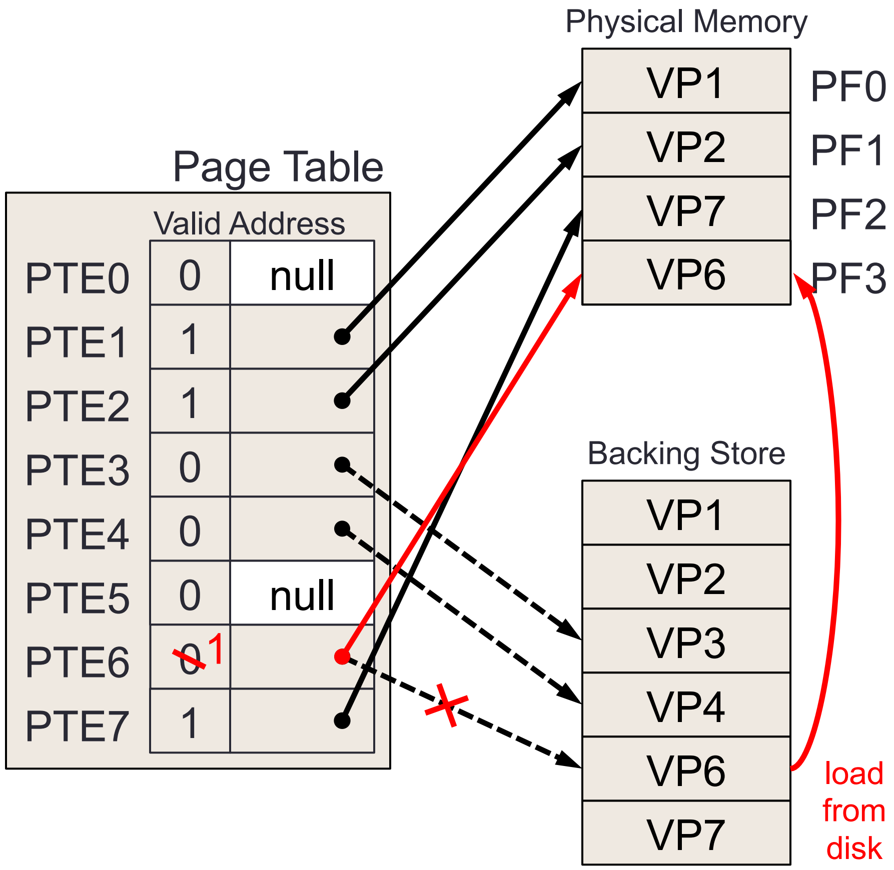
Virtual Memory Accesses (4)

- Kernel selects virtual page 4 as the victim page…
  - If VP4 has been changed, kernel writes it back to the disk
- Now that virtual page 4 is no longer valid, the kernel updates the page table to reflect this



Virtual Memory Accesses (5)

- Now kernel can load virtual page 6 into physical page 3
  - Update PTE6 to point to physical page 3 in DRAM memory
- Finally, kernel returns from the page-fault handler
  - Since it's a fault, the CPU reruns the faulting instruction
- Program repeats the access to virtual page 6
  - This time, MMU finds that PTE6 is valid
  - MMU performs address translation, and retrieves the value from physical page 3



Page Table

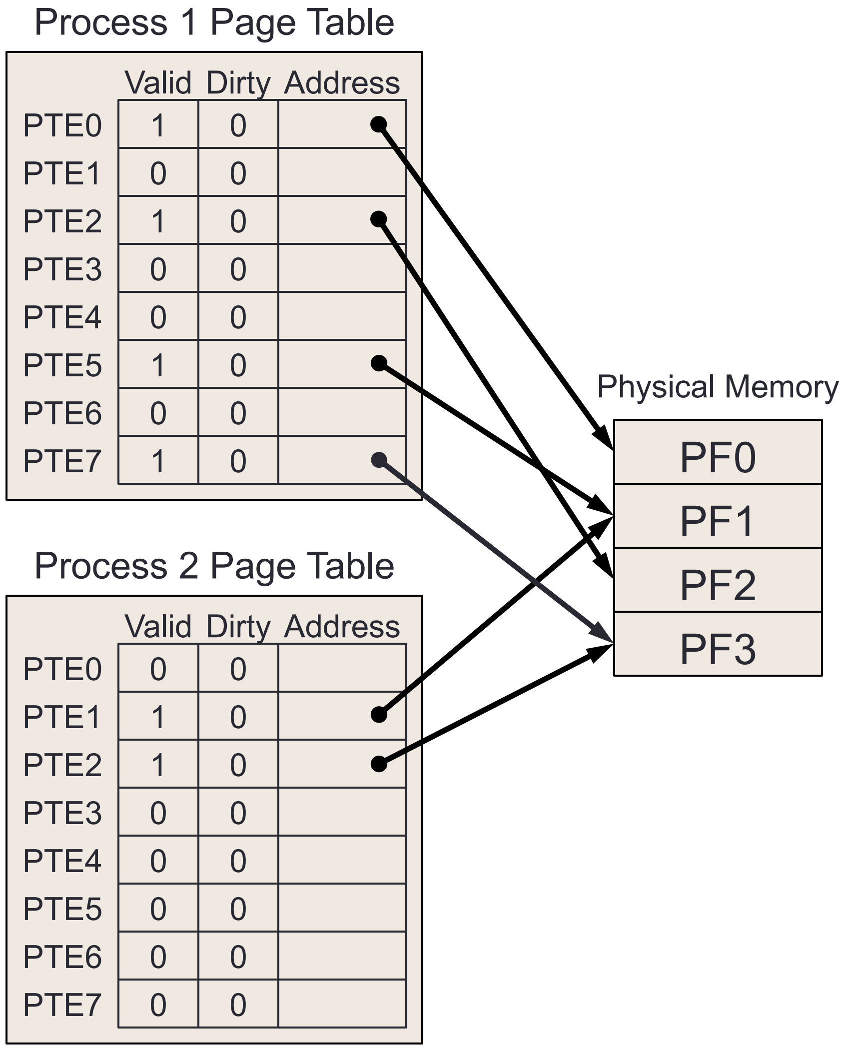
	Valid	Address
PTE0	0	null
PTE1	1	
PTE2	1	
PTE3	0	
PTE4	0	
PTE5	0	null
PTE6	~~0~~ 1	
PTE7	1	

Physical Memory

VP1	PF0
VP2	PF1
VP7	PF2
VP6	PF3

Page Tables and Dirty Flags

- **Page table dirty flags must be used with caution**
- Multiple virtual pages can map to one physical page-frame
  - e.g. shared memory used by multiple processes
  - e.g. pages mapped into kernel-space addresses, and also into a process' user-space addresses
- MMU only sets the dirty flag in the page-table entry that was used for the access
  - Other page-table entries that use the same physical page <u>are not</u> marked dirty



Process 1 Page Table

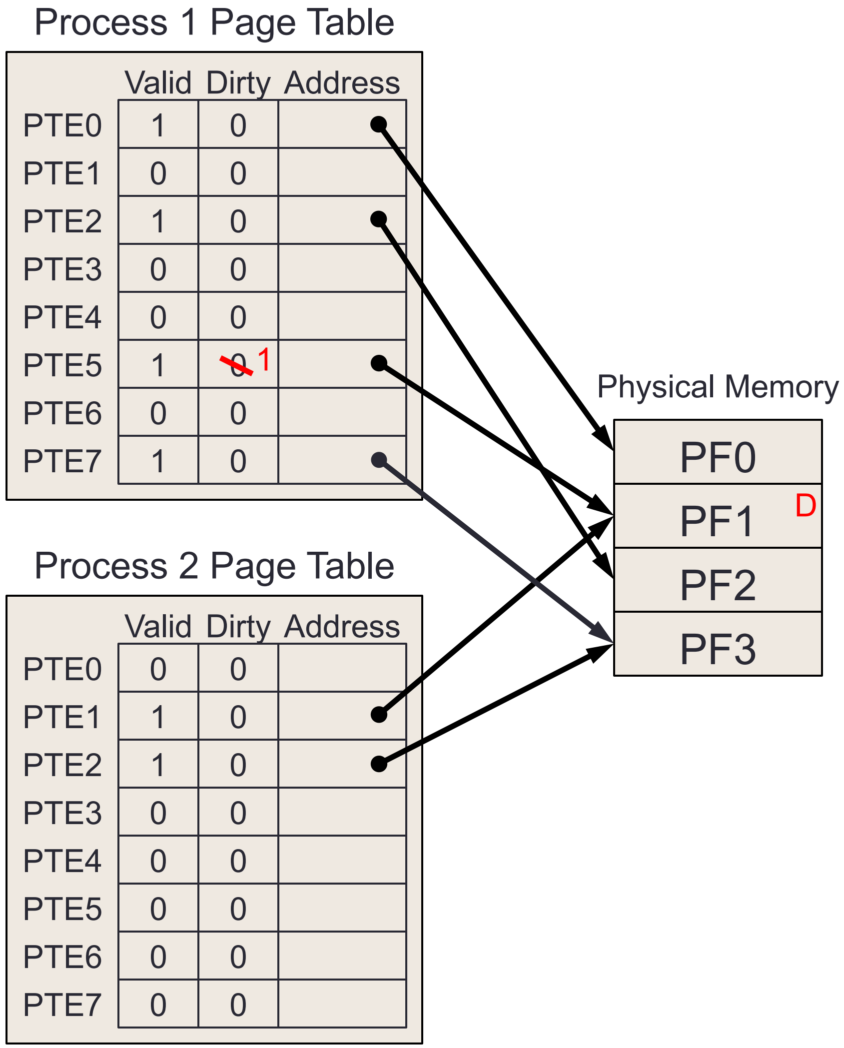
	Valid	Dirty	Address
PTE0	1	0	
PTE1	0	0	
PTE2	1	0	
PTE3	0	0	
PTE4	0	0	
PTE5	1	0	
PTE6	0	0	
PTE7	1	0	

Process 2 Page Table

	Valid	Dirty	Address
PTE0	0	0	
PTE1	1	0	
PTE2	1	0	
PTE3	0	0	
PTE4	0	0	
PTE5	0	0	
PTE6	0	0	
PTE7	0	0	

Page Tables and Dirty Flags (2)

- Example: Process 1 writes to virtual page 5
  - MMU translates this to PP1
  - Since it’s a write, the dirty bit is set
- Later, kernel decides to page out Process 2’s virtual page 1
  - Use frame 1 for a different page
- Problem:
  - Virtual page is dirty from earlier write, but Process 2’s PTE1 doesn’t show the page as dirty
- Kernel must handle **aliases** in the various page tables
  - Multiple pages referring to a single physical frame



Process 1 Page Table

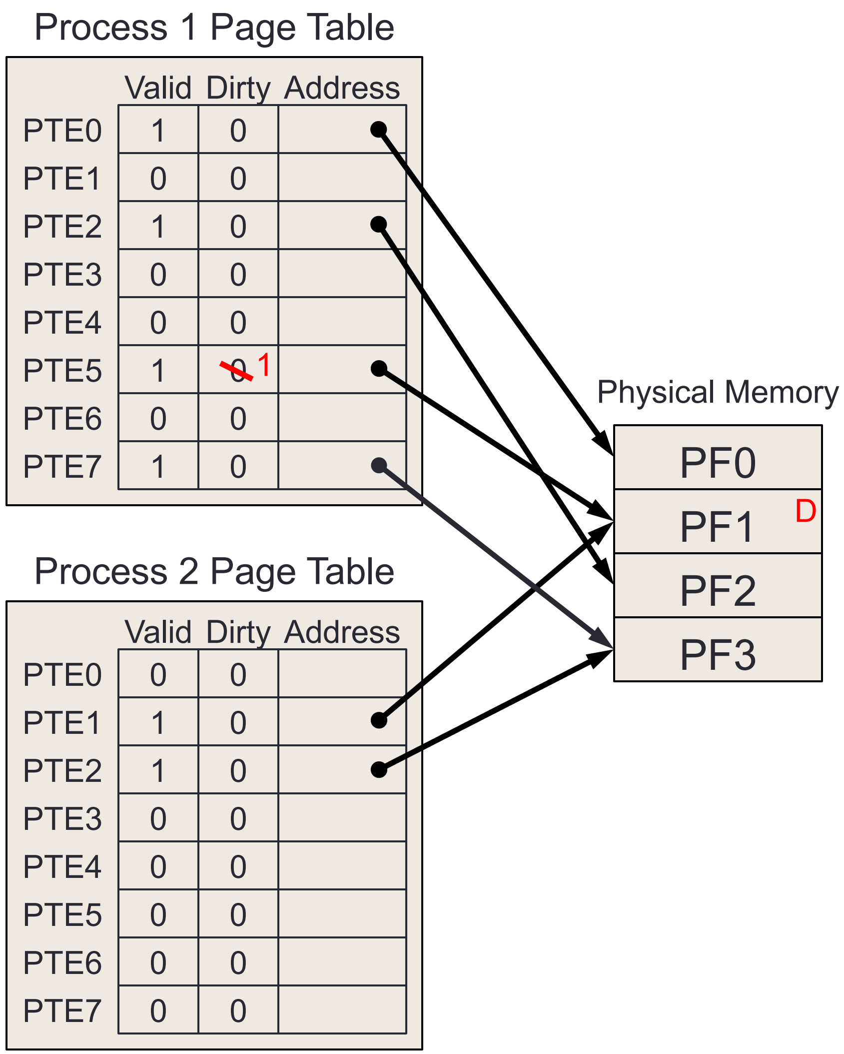
	Valid	Dirty	Address
PTE0	1	0	
PTE1	0	0	
PTE2	1	0	
PTE3	0	0	
PTE4	0	0	
PTE5	1	~~0~~ 1	
PTE6	0	0	
PTE7	1	0	

Process 2 Page Table

	Valid	Dirty	Address
PTE0	0	0	
PTE1	1	0	
PTE2	1	0	
PTE3	0	0	
PTE4	0	0	
PTE5	0	0	
PTE6	0	0	
PTE7	0	0	

Page Tables and Dirty Flags (3)

- Generally, kernel must check all PTEs for a given page before evicting that page
- Kernel must do this anyway:
  - If a page is evicted, all PTEs that referenced the page must be set to invalid…
- Can make this faster with kernel data structures
  - e.g. a frame table that records this information
  - Record areas that are actually shared between processes
- When a page is evicted:
  - If page is in a shared area, use additional kernel data to check and modify all relevant PTEs



Process 1 Page Table

	Valid	Dirty	Address
PTE0	1	0	
PTE1	0	0	
PTE2	1	0	
PTE3	0	0	
PTE4	0	0	
PTE5	1	~~0~~ 1	
PTE6	0	0	
PTE7	1	0	

Process 2 Page Table

	Valid	Dirty	Address
PTE0	0	0	
PTE1	1	0	
PTE2	1	0	
PTE3	0	0	
PTE4	0	0	
PTE5	0	0	
PTE6	0	0	
PTE7	0	0	

Swapping and Paging

- Paging allows the kernel to move parts of a process into and out of memory
  - Much better than standard swapping, where entire processes are moved between memory and the backing store
- In fact, the kernel can implement a **demand paging** policy
  - Only swap (or allocate) a virtual page into physical memory when it is actually used
- Example: running a program stored on disk
  - Kernel sets up a page table that references the program's binary…
  - But, none of the program's pages are actually in virtual memory! All pages are still on disk.
  - When the process begins running, it immediately triggers a page fault
    - Kernel loads the first page of the program's code into memory
  - As the program runs, accesses to new pages cause page faults
    - Those pages are loaded into memory as they are required
  - Only the parts of the program that *actually run* are loaded into memory

Demand Paging (2)

- Another example: managing a process' memory heap
- Initially, the kernel sets up a memory area for the process' heap, but all virtual pages in that memory area are initially invalid
- As the process actually interacts with the heap, the kernel allocates virtual pages to back the heap memory area
  - e.g. as program allocates space, manipulates data, deallocates space
- If process' heap size must be increased, kernel repeats this task
  - Expand the memory area, but don't allocate virtual pages until the process actually tries to use the memory area
- The kernel will only allocate as many virtual pages to the heap as are actually required by the program

Demand Paging (3)

- At some point, the kernel won’t have frames available to hold a virtual page
- When a new frame is needed, the kernel must choose a victim page to evict
  - Victim page is chosen according to the **page replacement policy** of the system
  - This page is written to a swap area on disk, and the frame is used for the new virtual page
- A kernel can implement **pure demand paging**
  - Only ever allocate/load virtual pages when they are required
  - Only ever evict pages when the system is out of physical frames
- Usually, kernels manage memory more actively than this
  - Increase application responsiveness by **prefetching** virtual pages, reduce amount of dirty data cached in memory, etc., etc.

Copy-On-Write

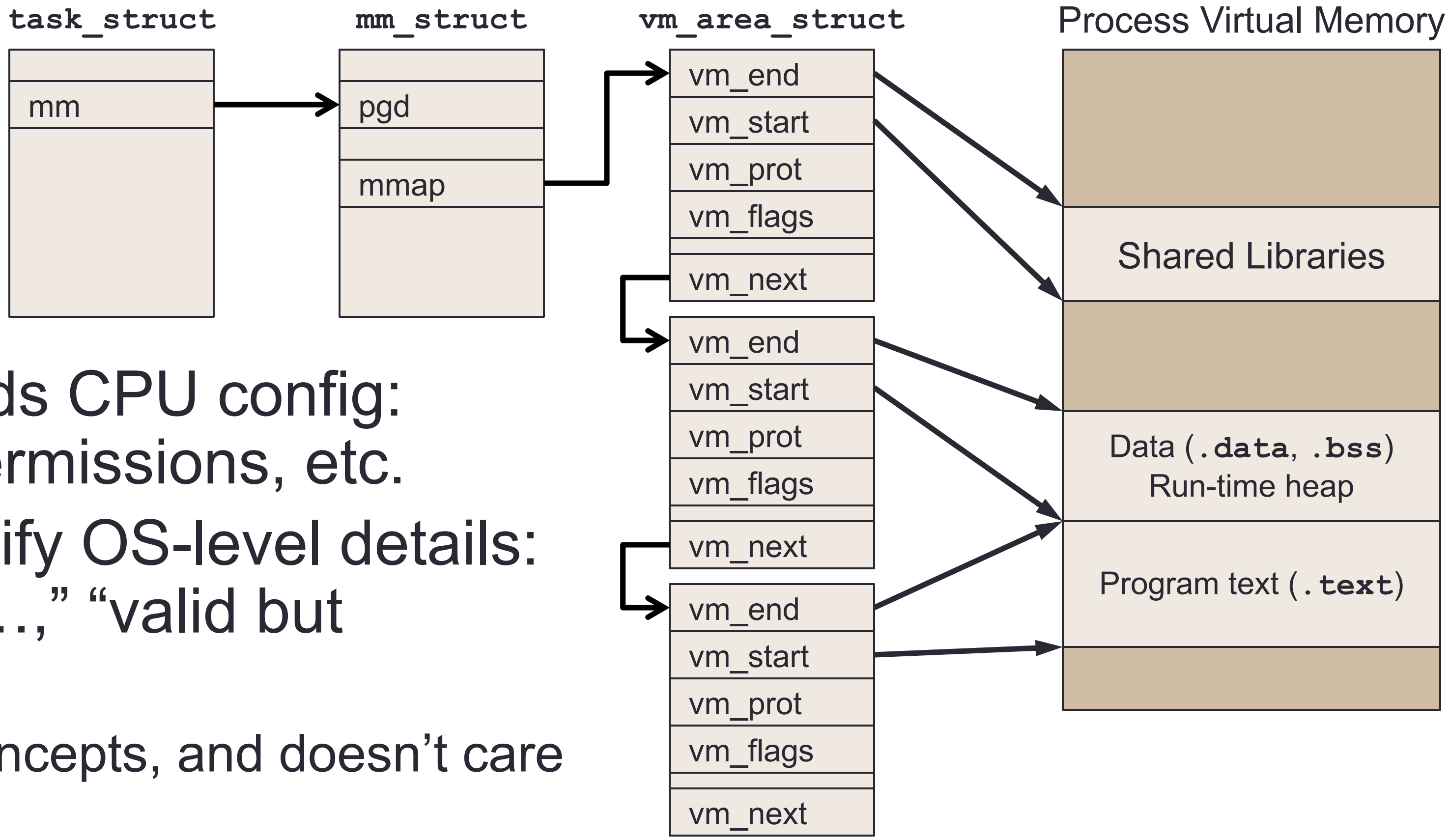
- A similar technique is used to give processes the illusion of independent copies of specific pages
- Example: forking a process on UNIX
  - The parent and child processes are identical copies of each other, but their state is isolated from each other
  - Parent and child execution begins to diverge, and their state starts to diverge as well
- The kernel can use a **copy-on-write** technique for forking
  - When parent and child process split, share all pages between them
  - As long as pages are shared, they cannot be written to; otherwise each process would see the other's changes
  - The kernel sets all shared pages to be read-only in the PTEs
  - The MMU enforces this constraint by raising a fault on writes

Copy-On-Write (2)

- When a process tries to write to a read-only page, the MMU triggers a fault
    - The kernel determines that the page is in a copy-on-write area
    - If more than one process is still sharing this page, kernel makes a private copy for the writer
        - (If only one process is using the page then a copy step is not needed)
    - Kernel updates the process' page table to point to the private page
    - Kernel returns to the faulting write-instruction, which now succeeds
- This mechanism *greatly* improves **`fork()`** performance
    - Instead of duplicating all the virtual pages, only the page table needs to be duplicated (and updated to set up copy-on-write)

Memory Area Descriptors

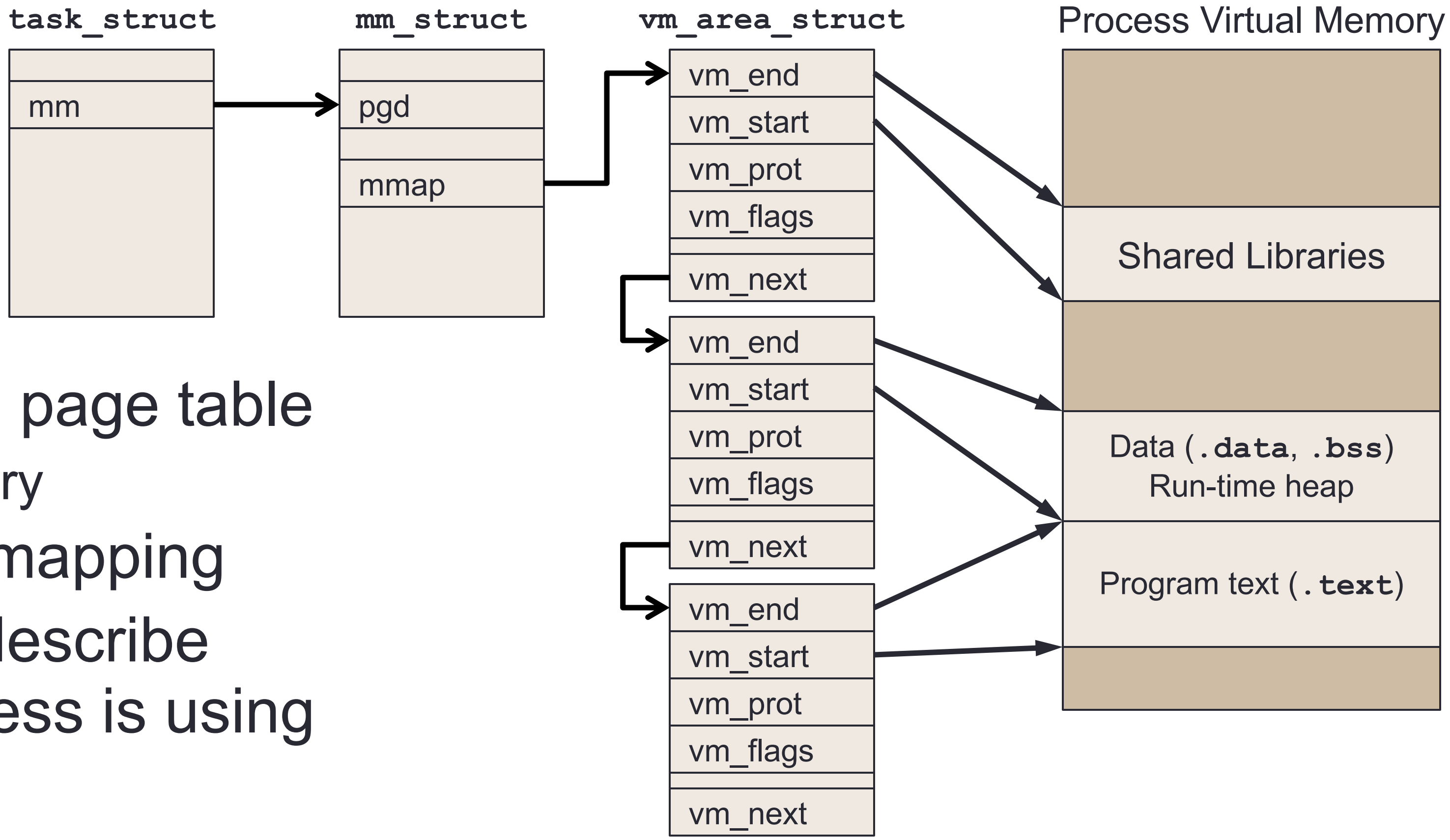
- All of these virtual memory features require kernels to record details beyond what the page table holds
  - Structure is sometimes called a **supplemental page table**



- MMU page-table structure holds CPU config: valid bit, read/write/execute permissions, etc.
- Memory area descriptors specify OS-level details: “copy-on-write,” “shared with …,” “valid but unallocated,” etc.
  - CPU doesn’t understand these concepts, and doesn’t care

Memory Area Descriptors (2)

- This example of descriptors is from the Linux 2.6 kernel
  - **`task_struct`** is the Linux process (thread) control block



`task_struct`

mm

`mm_struct`

pgd

mmap

`vm_area_struct`

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_end
vm_start
vm_prot
vm_flags
vm_next

Process Virtual Memory

Shared Libraries

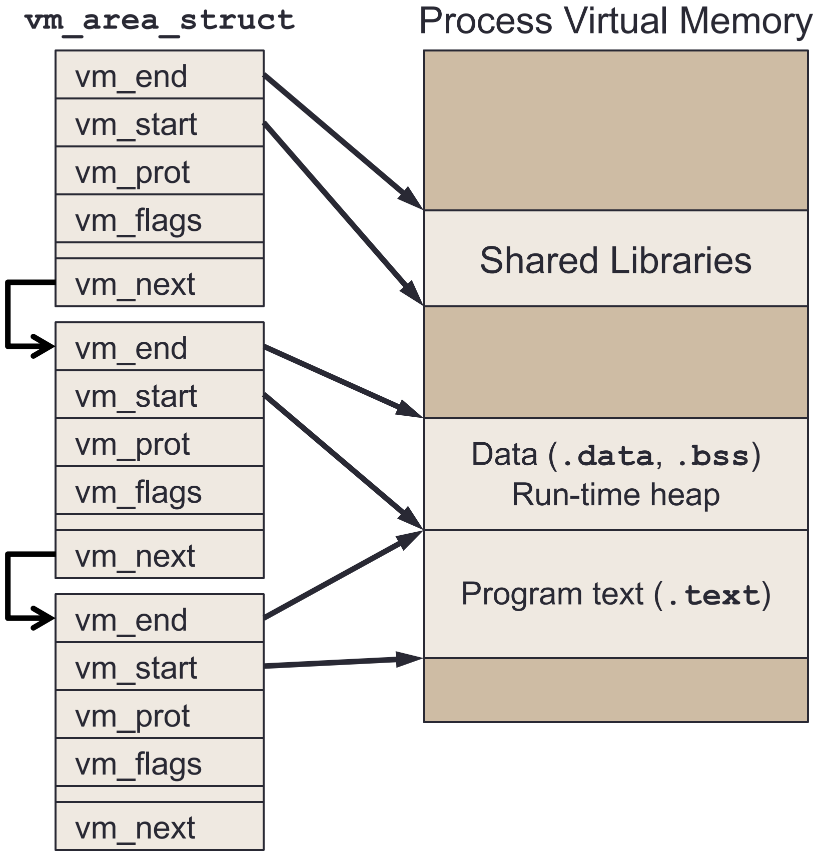
Data (**`.data`**, **`.bss`**)
Run-time heap

Program text (**`.text`**)

- **`pgd`** is the process' CPU/MMU page table
  - "pgd" = pointer to the page directory
- **`mmap`** is the process' memory mapping
- **`vm_area_struct`** elements describe virtual memory areas the process is using

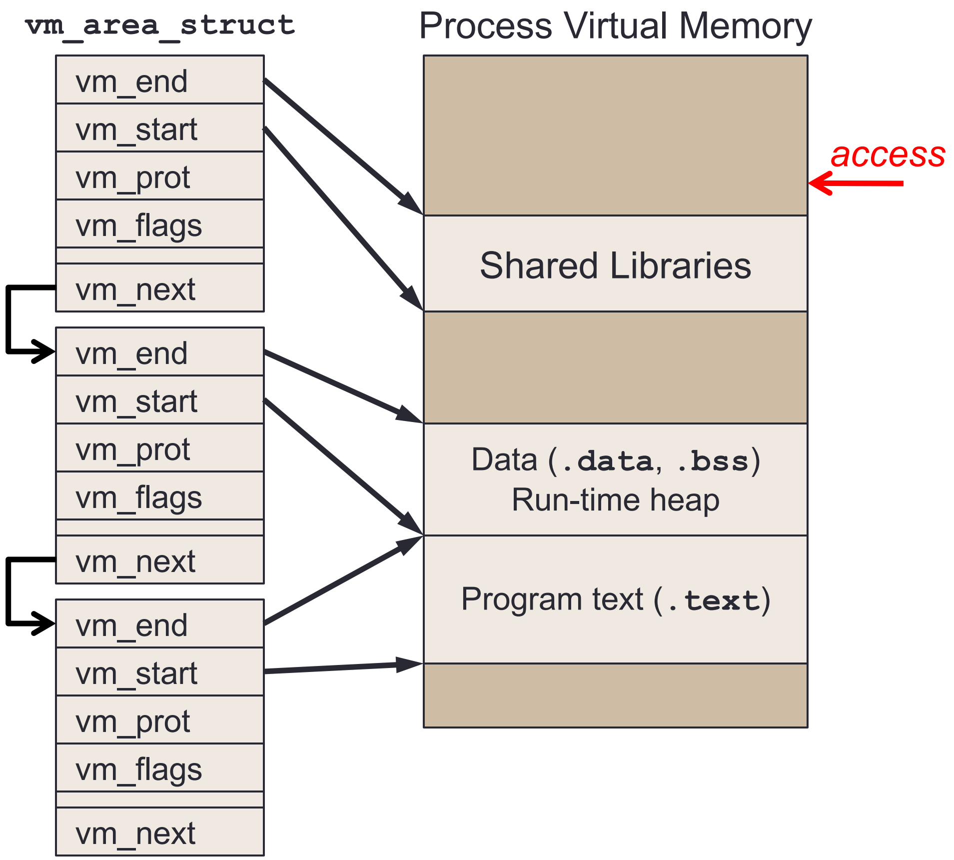
Memory Area Descriptors (3)

- **`vm_area_struct`** specifies details of each memory area
  - **`vm_start`**, **`vm_end`** specify the extent of the memory area
  - **`vm_prot`** specifies the read/write permissions for the memory area
  - **`vm_flags`** specifies whether memory area is shared among processes, or private
- Normal memory accesses:
  - (i.e. virtual page is in memory, and the operation is allowed)
  - No intervention is needed from the kernel…
  - CPU and MMU handle these accesses themselves



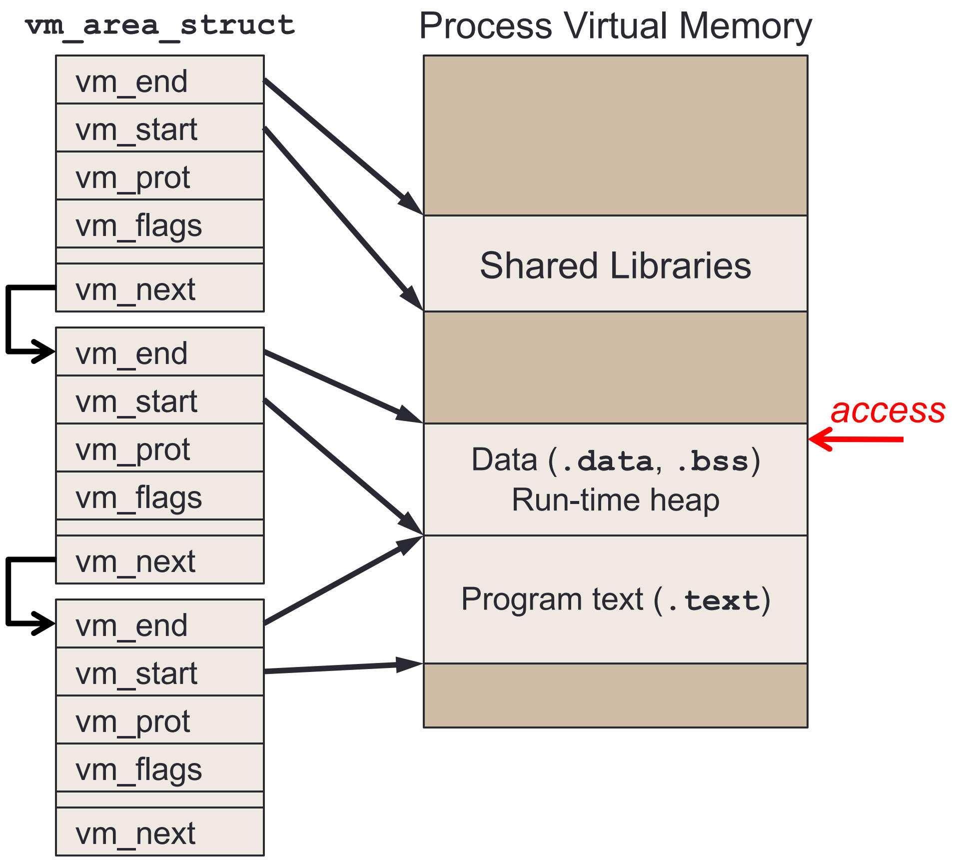
Handling Page Faults

- When a fault occurs, the kernel must resolve the situation
  - Process' **`vm_area_struct`** list tells kernel how to handle the fault
- If MMU raises a page fault:
  - Page isn't currently in the process' address space
- Kernel checks all areas to see if the address is valid
  - Does it fall within some **`vm_start`** and **`vm_end`**?
- If address isn't valid, kernel sends an appropriate signal to the process
  - e.g. **`SIGSEGV`**; usually causes the process to terminate



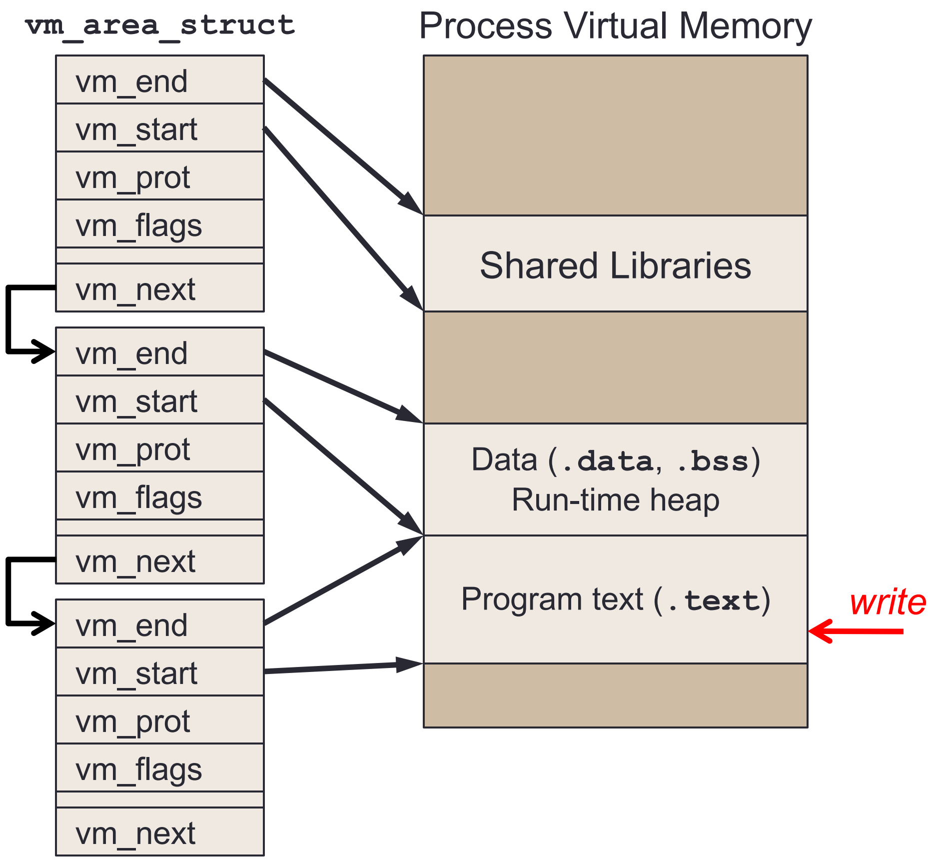
Handling Page Faults (2)

- At this point, the page is either swapped out to storage, or the page hasn't yet been allocated by the kernel
- If the page is swapped out, kernel initiates a page-load, then switches to another process
- If page isn't allocated yet, the kernel allocates a new page to the process
  - New page is filled with zeros to prevent leaking data between processes



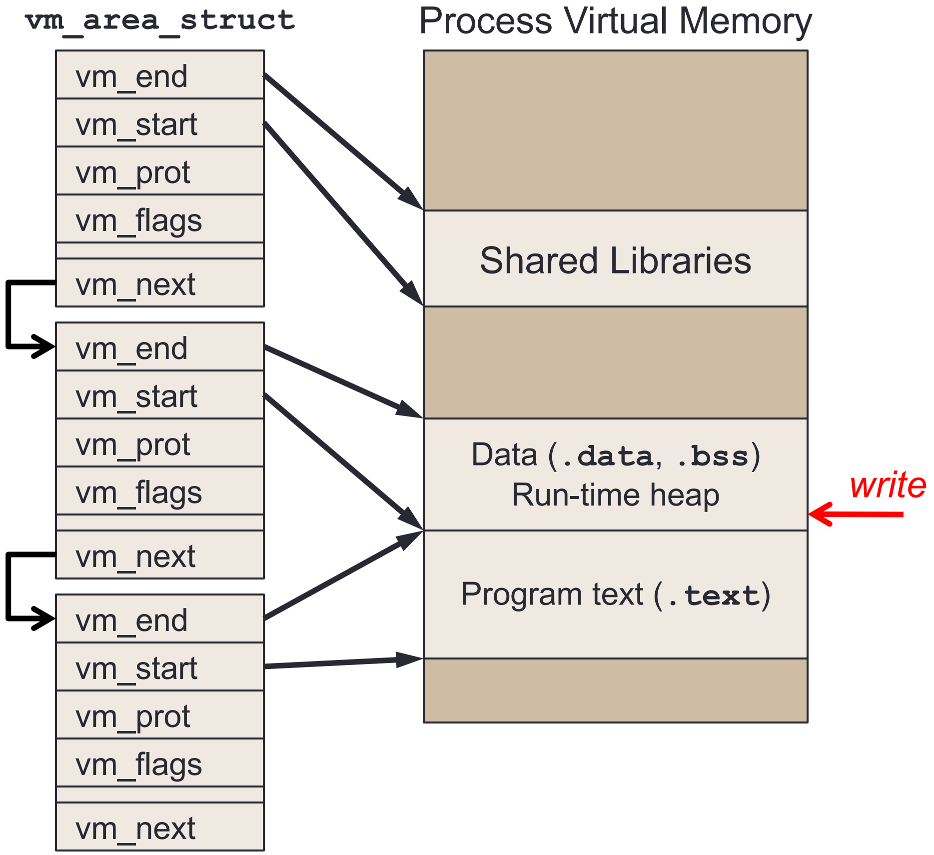
Handling Protection Faults

- If MMU raises a general protection fault:
  - Process tried to do something that is prohibited by the page table
  - e.g. write to a read-only page
- Kernel checks to see how the virtual memory area is configured
  - Is it a copy-on-write area?
- If memory area doesn't allow the operation, again a signal is sent to the process
  - e.g. **`SIGSEGV`**; usually causes process to terminate



Handling Protection Faults (2)

- If the memory area does allow the operation, the kernel carries it out
- Example: copy-on-write
  - If necessary, duplicate the faulting page
  - Update the process' page table:
    - Point the entry to the new frame containing the copy
    - Mark the page as read-write



Next Time

- More kernel virtual memory management details