



CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2014

Lecture 24

LAST TIME

- Extended virtual memory concept to be a cache of memory stored on disk
 - DRAM becomes L4 cache of data stored on L5 disk
- Extend page table entries with more details
 - Entries have a *valid* (IA32: “present”) flag specifying if the page is in memory, and if not, where it resides
 - Also permission flags, e.g. “read/write,” “supervisor”
- Requires hardware and software support:
 - CPU performs address translation in hardware to make it as fast as possible
 - CPU raises page fault and general protection fault exceptions when it requires the kernel’s intervention
 - Operating system handles situations where pages must be moved into and out of memory

LAST TIME (2)

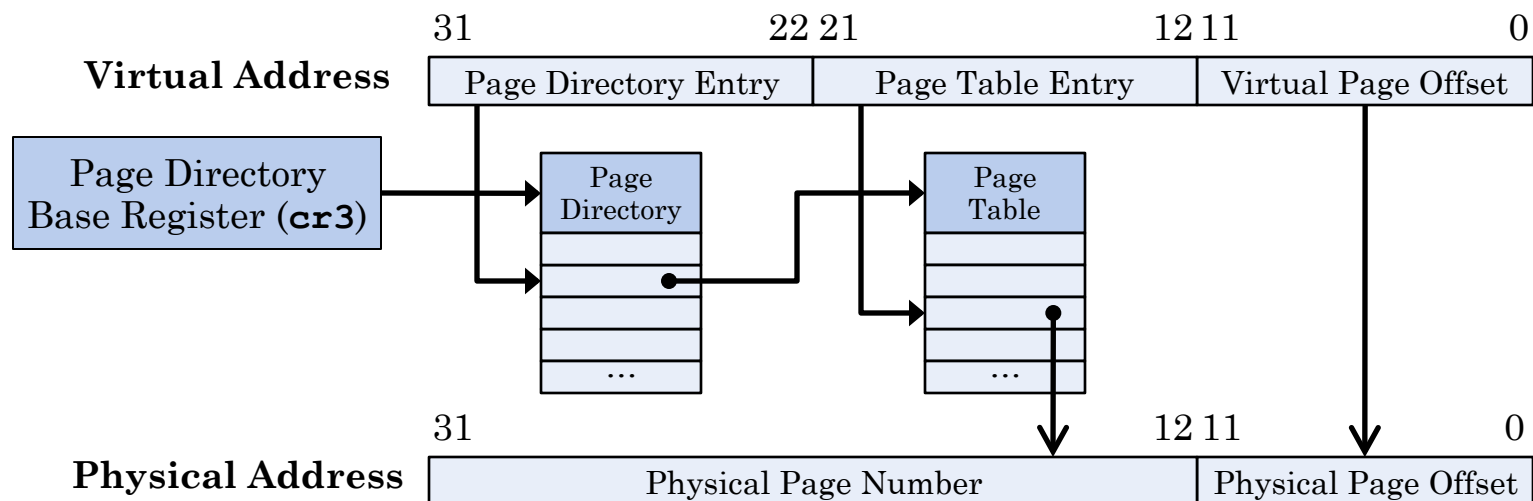
- Can now implement many useful features!
- Isolate address spaces of different processes
- Perform fast context-switches by changing the page table that the MMU uses
- Share memory regions between processes, such as shared libraries, kernel code, working areas
- Memory-map disk files into virtual memory, to load programs, and to perform fast and easy IO
- Set permissions on memory pages to make some pages read-only, or inaccessible by user code

IA32 VIRTUAL MEMORY SUPPORT

- Intel Pentium-family processors provide hardware support for virtual memory
- Virtual and physical address spaces are 32 bits
- Pages are 4KB in size ($2^{12} = 4096$)
 - Pages are identified by topmost 20 bits in address
 - Offset within page specified by low 12 bits in address
- Pentium-family processors implement a two-level page table hierarchy
- Level 1 is the *page directory*
 - Entries in the page directory refer to page tables, as long as the page table is not empty
- Level 2 contains *page tables*
 - Entries map virtual pages to physical pages

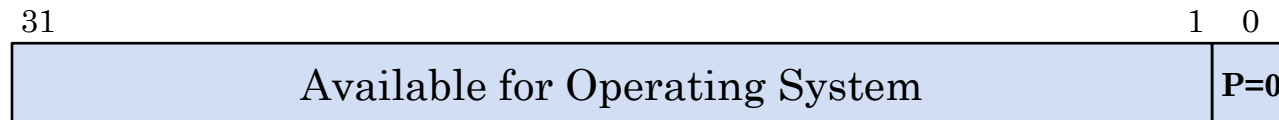
PENTIUM-FAMILY PAGE DIRECTORIES

- Each process has its own page directory
 - Each process has its own virtual address space, isolated from all other processes
 - Page directory also maps some kernel code and shared library code into the process' address space
- Current page directory is specified by the Page Directory Base Register
 - On IA32, this is **%cr3**, or Control Register 3
 - *Only the kernel can change this control register!*



PAGE DIRECTORY/TABLE ENTRIES

- IA32 page directory and table entries are 32 bits
 - 20 bits used to specify physical address of either a page table, or a virtual memory page
 - Other bits contain additional details about the entry
- Bit 0 (least-significant bit) is the Present bit
 - (i.e. the *valid* bit from last lecture)
 - When 1, the referenced page is cached in memory
 - When 0, the referenced page is not in memory (e.g. page is stored on disk)
- When Present = 0, all other bits are available for the kernel to use



- Specifies location on disk of where the page is stored

PAGE DIRECTORY/TABLE ENTRIES (2)

- When Present bit is 1, page directory and page table entries contain several bookkeeping values
- Page directory entry:



- Page table entry:



- Very similar contents for both kinds of entries

PAGE DIRECTORY/TABLE ENTRIES (3)



- Bits 1 and 2 specify access permissions
 - R/W = 1 is read/write, R/W = 0 is read-only
 - U/S = 1 is user access, U/S = 0 is kernel access only
- What other permission might we want?
 - An Execute permission!
- *Dramatically* reduces potential for buffer-overflow exploits!
 - Set stack and data pages to not be executable
 - Set code pages to be executable and read-only
- More recent IA-32 and Intel 64 CPUs have ability to disable execution for an entire page-directory entry

PAGE DIRECTORY/TABLE ENTRIES (4)

31	12	11	7	6	5	4	3	2	1	0
Page Base Address		...		D	A	PCD	PWT	U/S	R/W	P=1

- Bits 3 and 4 specify caching policies for the page
 - PWT specifies write-through or write-back
 - PCD specifies whether cache is enabled or disabled
- Some peripherals are mapped directly into the computer's memory address space
 - Technique is called *memory-mapped I/O*
 - CPU interacts with the peripheral by reading and writing specific memory locations
 - Memory addresses read/write directly to the I/O device
 - These addresses are called *I/O ports*
- Definitely don't want to cache the memory page in these cases!

PAGE DIRECTORY/TABLE ENTRIES (5)

31	12	11	7	6	5	4	3	2	1	0
Page Base Address		...	D	A	PCD	PWT	U/S	R/W	P=1	

- Bit 5 is the Accessed bit
 - MMU sets this to 1 when the page is read or written
 - Kernel is responsible for clearing this bit
- Accessed bit used to track what pages have been used
 - Helps kernel decide which page to evict when it needs to free up space in physical memory
- Bit 6 is the Dirty bit
 - Only in page table entries, not page directory entries!
 - MMU sets this to 1 when the page is written to
- Dirty bit allows kernel to know when a victim page must be written back to the disk before it is evicted
 - Kernel is responsible for handling and clearing this bit

PAGE DIRECTORY/TABLE ENTRIES (6)

31	12	11	7	6	5	4	3	2	1	0
Page Base Address		...		D	A	PCD	PWT	U/S	R/W	P=1

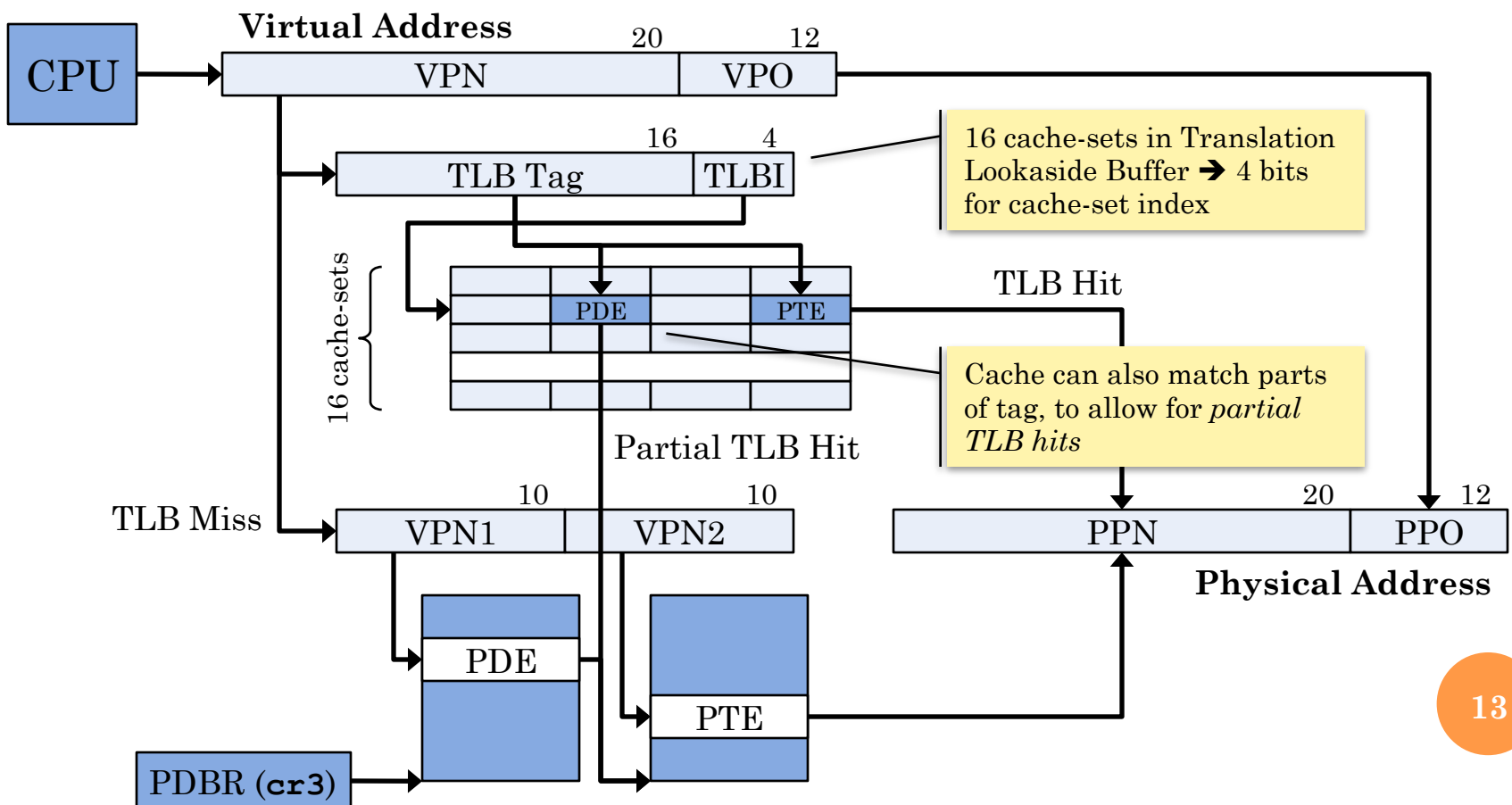
- Processor causes faults in certain situations
- If Present = 0 when a page is accessed, the CPU raises a *page-fault* exception
 - Kernel page-fault handler can load the page into memory if it's on disk
 - Or, if the page is unallocated, generate an error
- If Read/Write or User/Supervisor bits prohibit an access, CPU raises a *general protection fault*
 - Kernel general protection fault handler can respond in various ways, but typically process is terminated

IA32 ADDRESS TRANSLATION AND TLBS

- Page directory and page tables are stored in DRAM main memory
 - Worst case: 50-100ns access penalty
 - If needed block is in L1 cache, 1-3 clock hit-time
- CPU includes a Translation Lookaside Buffer (TLB) to eliminate even this lookup penalty
 - A hardware cache with same design as SRAM caches
- IA32 family processors:
 - TLB is 4-way set-associative cache with 16 cache sets
 - Input to TLB cache is the virtual page number
 - Each cache line holds a page table entry, including the physical page number

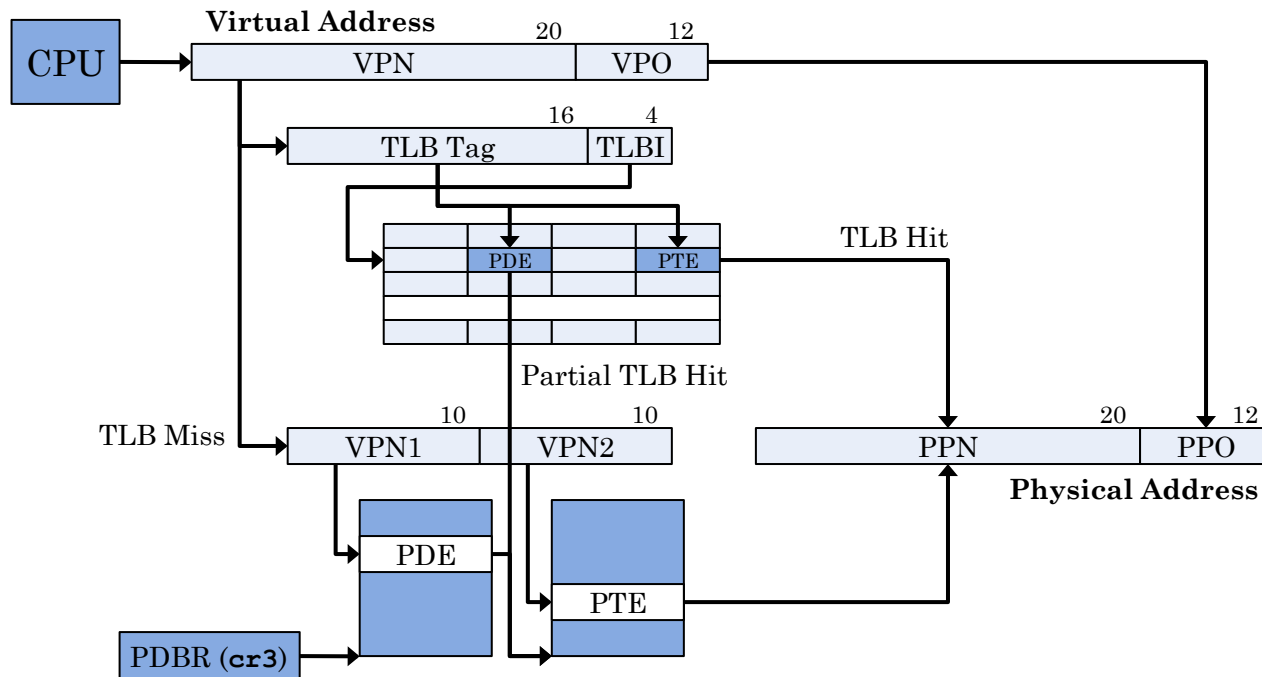
IA32 ADDRESS TRANSLATION, TLBs (2)

- Address translation logic:



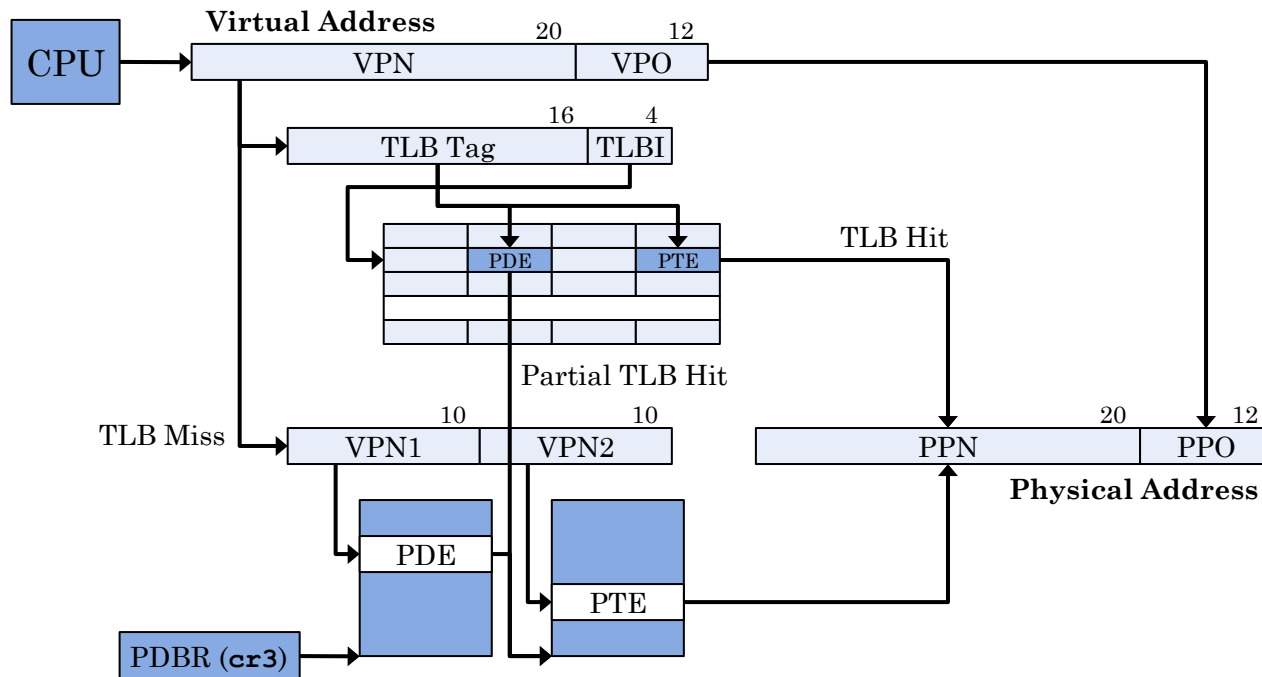
IA32 ADDRESS TRANSLATION, TLBs (3)

- In case of a TLB miss:
 - Virtual page number is broken into an index into the page directory, and an index into the page table
 - Incurs full lookup penalty, but the TLB cache is also updated with the results of the lookup



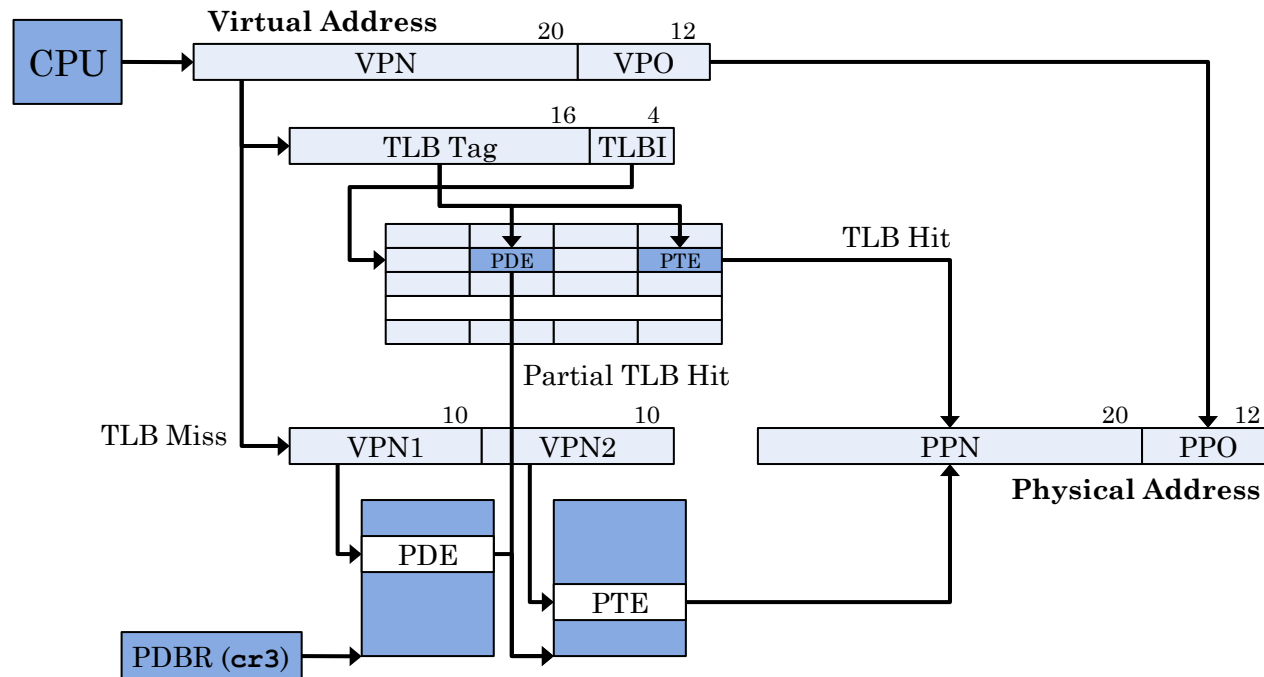
IA32 ADDRESS TRANSLATION, TLBs (4)

- Ideally, we want a TLB hit:
 - Virtual page number is broken into a tag and a cache-set index (TLBI), as usual
 - If TLB cache line contains page table entry (PTE), use this for the physical page number (PPN)



IA32 ADDRESS TRANSLATION, TLBs (5)

- Sometimes, we get a *partial TLB hit*
 - The page directory entry (PDE) is present in TLB, but not the page table entry
 - Use PDE to look up physical page number from specified page table, and cache result back into TLB

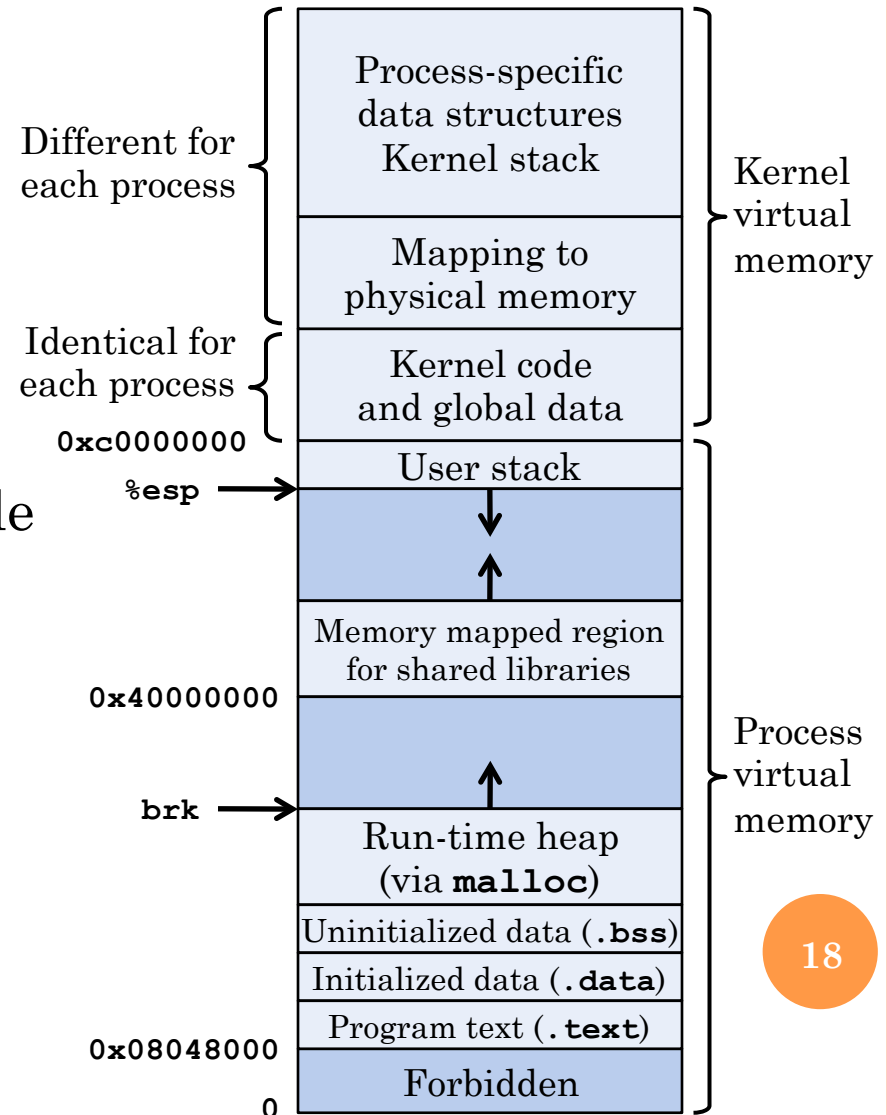


KERNEL AND VIRTUAL MEMORY SYSTEM

- The kernel plays an important role in the virtual memory system
 - Manages the page directories and page tables of running processes
 - Handles page faults and general protection faults
- Each process has its own virtual address space
 - Each process has its own page directory that specifies the process' virtual memory layout
- On IA32, only the kernel can change the current page directory being used
 - Requires level 0 (highest) privilege
 - Page tables are also only updatable by the kernel

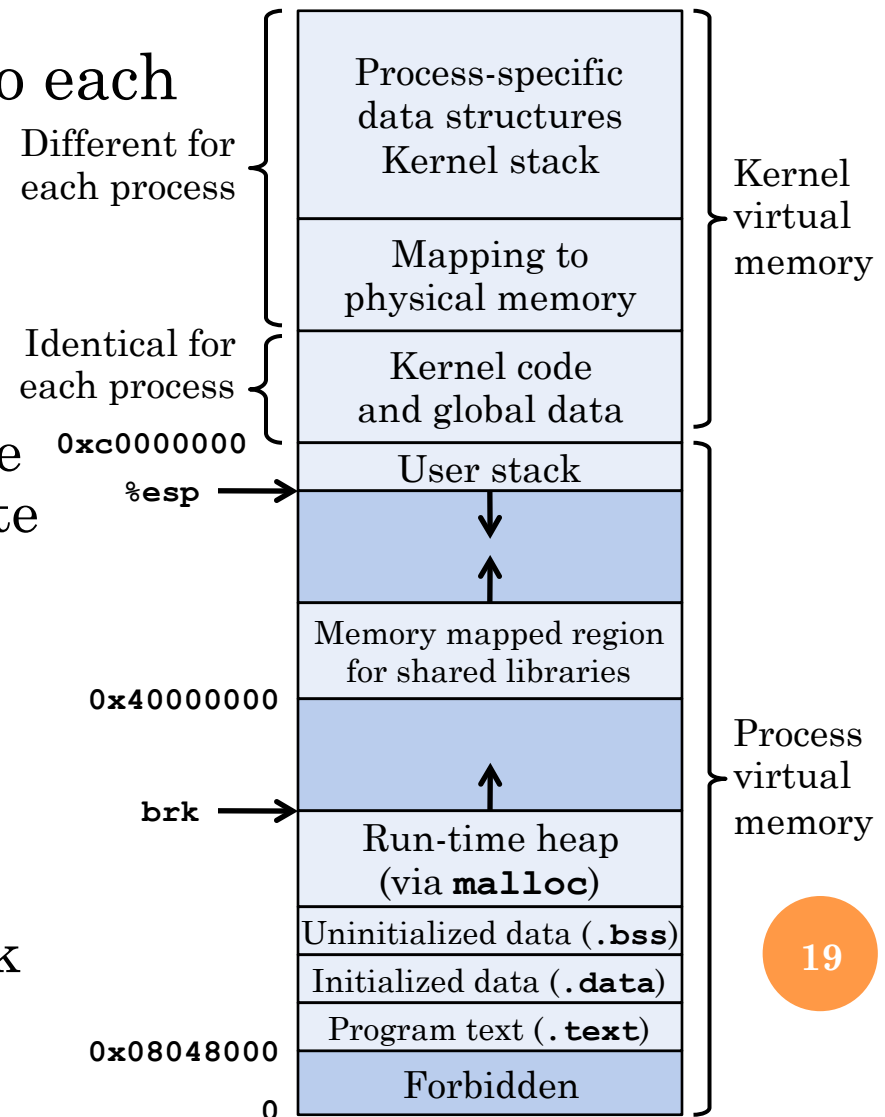
PROCESS MEMORY LAYOUT

- Each process has its own virtual address space
- Part of virtual address space is devoted to kernel
 - Region starting at address **0xc0000000**
 - This memory only accessible by the kernel
- Includes functionality and data structures necessary for all processes...
 - Simply map these physical pages into every process' virtual address space



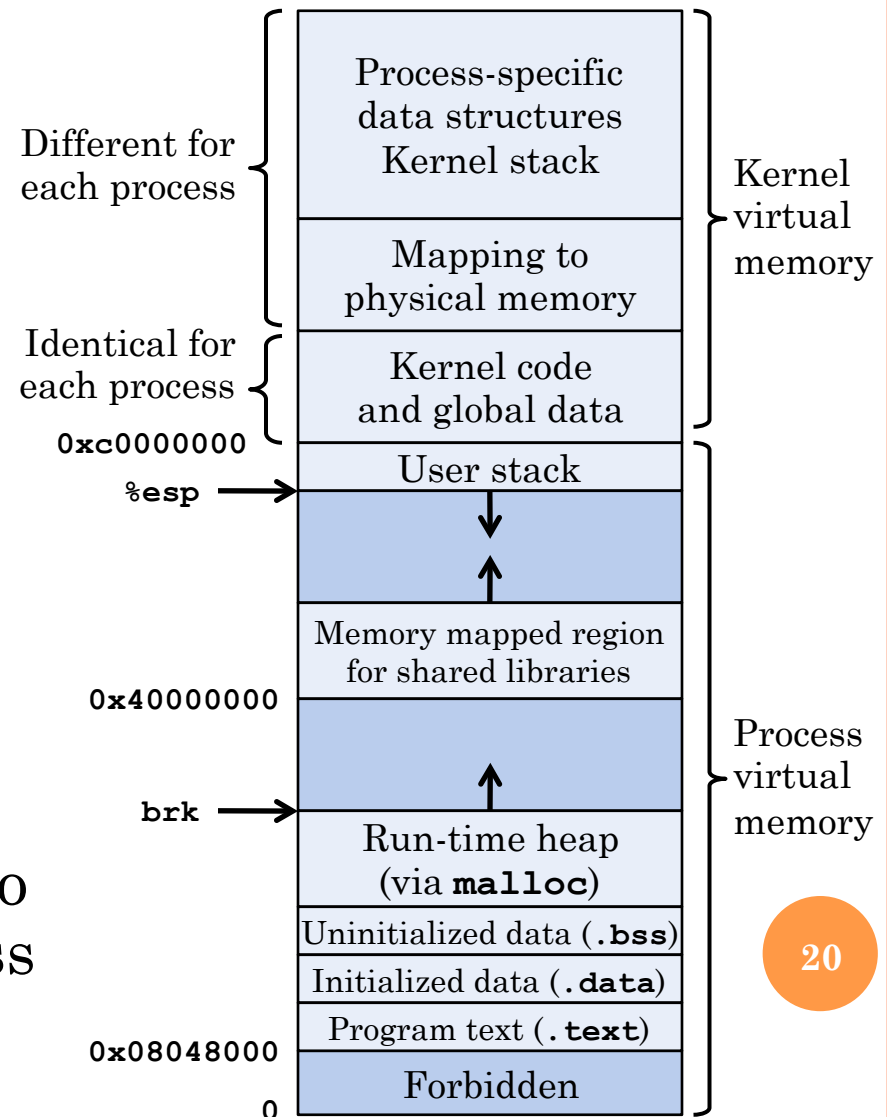
THE KERNEL AND SYSTEM CALLS

- Kernel code is mapped into each process' address space
 - Easy* to make system calls!
- Can call kernel code via **int 0x80** exception
 - Allows a change to privilege level 0, via an interrupt gate
- Problem: **int** is slow...
 - A generalized mechanism
 - Must get descriptor from Interrupt Descriptor Table
 - Perform privilege check
 - Set up/change the call stack
 - Jump to specified address



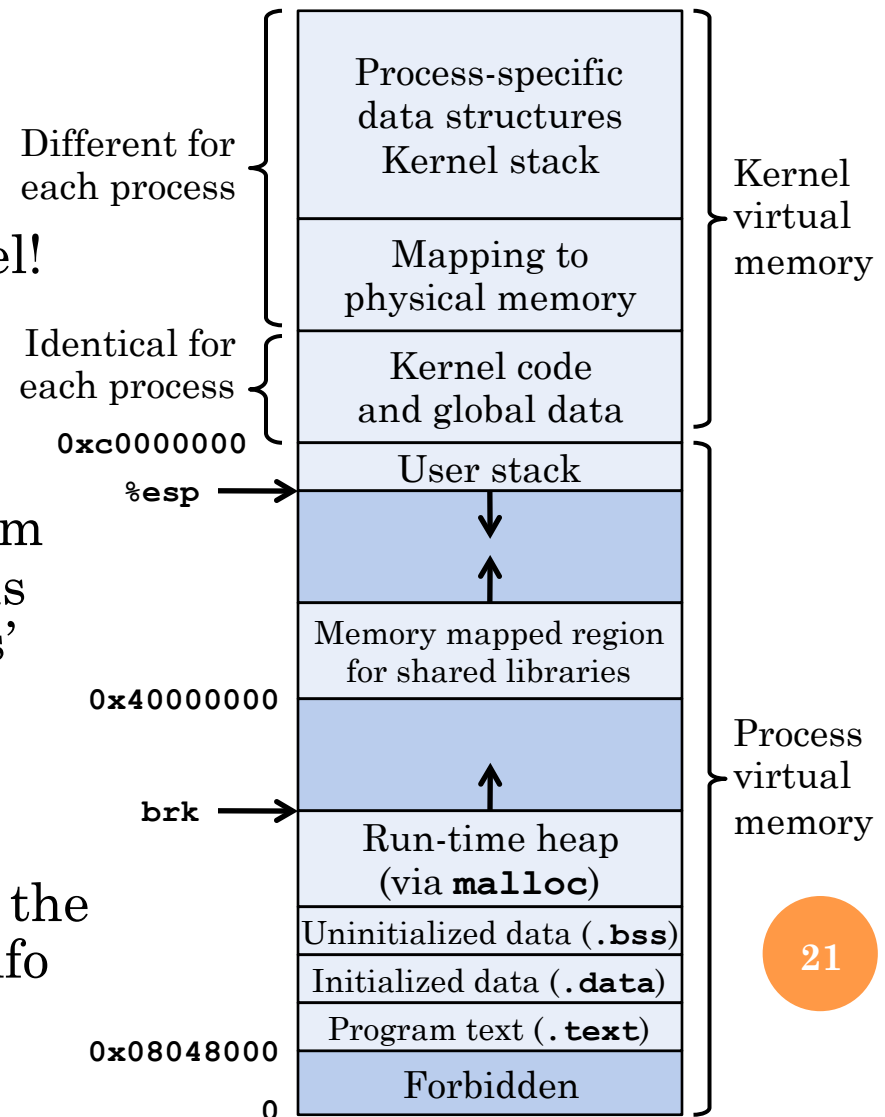
THE KERNEL AND SYSTEM CALLS (2)

- IA32 also provides Fast System Call support
 - **sysenter/sysexit**, or **syscall/sysret**
 - Uses special registers initialized by the kernel, specifying where to jump into the system code
 - Provides a fast and specific mechanism for moving to protection level 0
- Both of these mechanisms *require* some kernel code to be mapped into the address space of every process



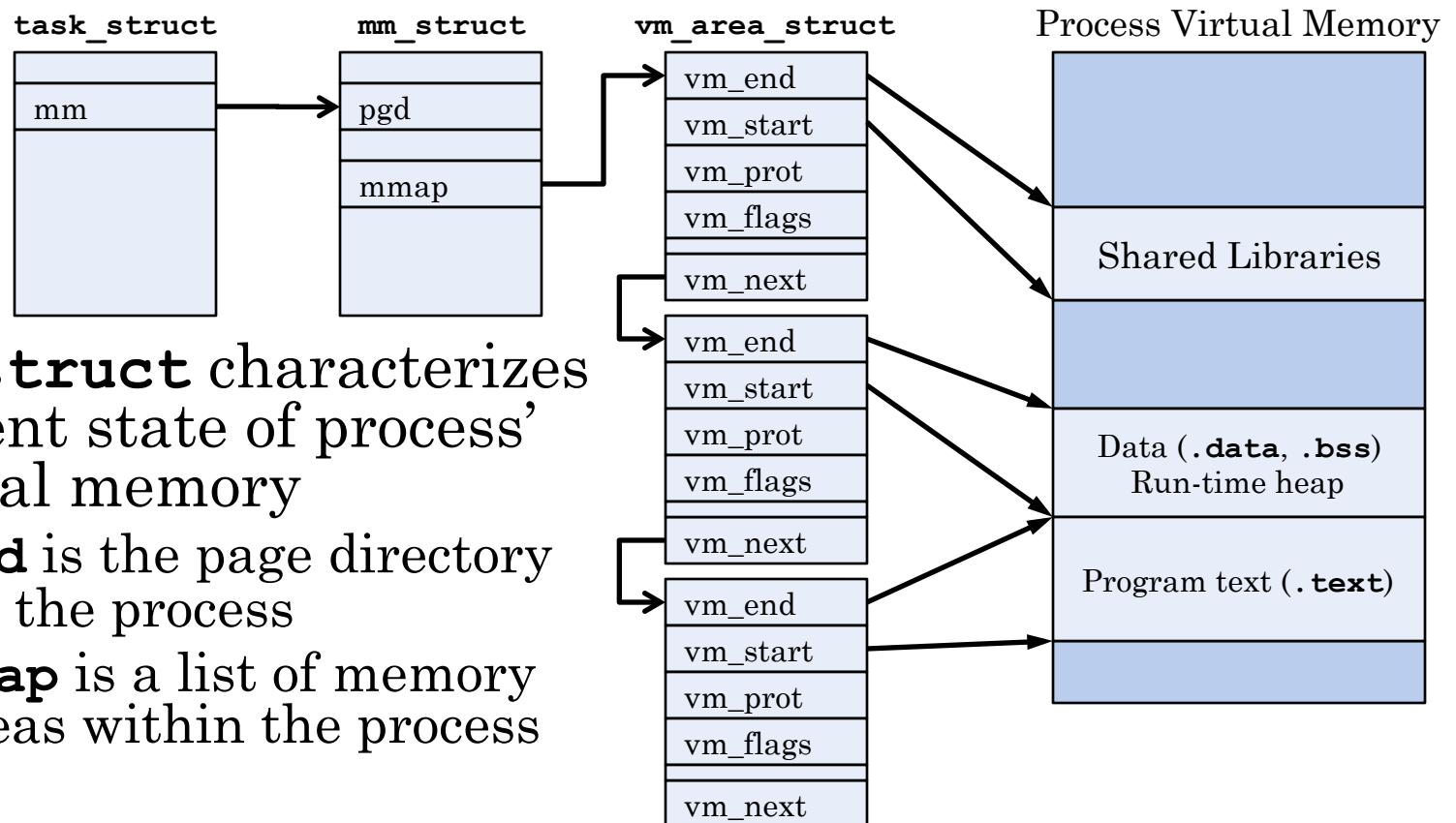
PROCESS-SPECIFIC KERNEL DATA

- Each process also includes process-specific structures, managed by the kernel
 - Also accessible only by kernel!
- Kernel stack:
 - Every protection level has its own stack...
 - When a process makes system calls, the kernel-stack used is also only within that process' address space
- Several other data structures are also managed per-process
 - e.g. page directory/tables for the process, memory mapping info



PROCESS VIRTUAL MEMORY AREAS

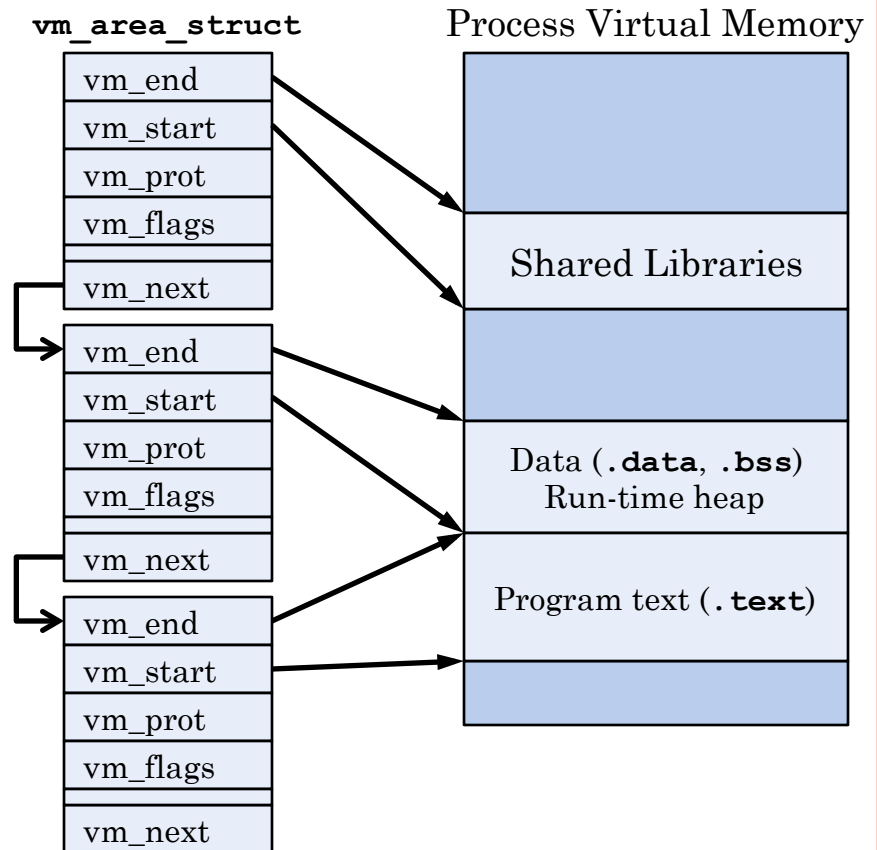
- Kernel manages several data structures to track virtual memory regions within each process
 - Regions are called “areas” or “segments”
 - (Not related to old 8086 segmented memory model!)



- mm_struct** characterizes current state of process' virtual memory
 - pgd** is the page directory for the process
 - mmap** is a list of memory areas within the process

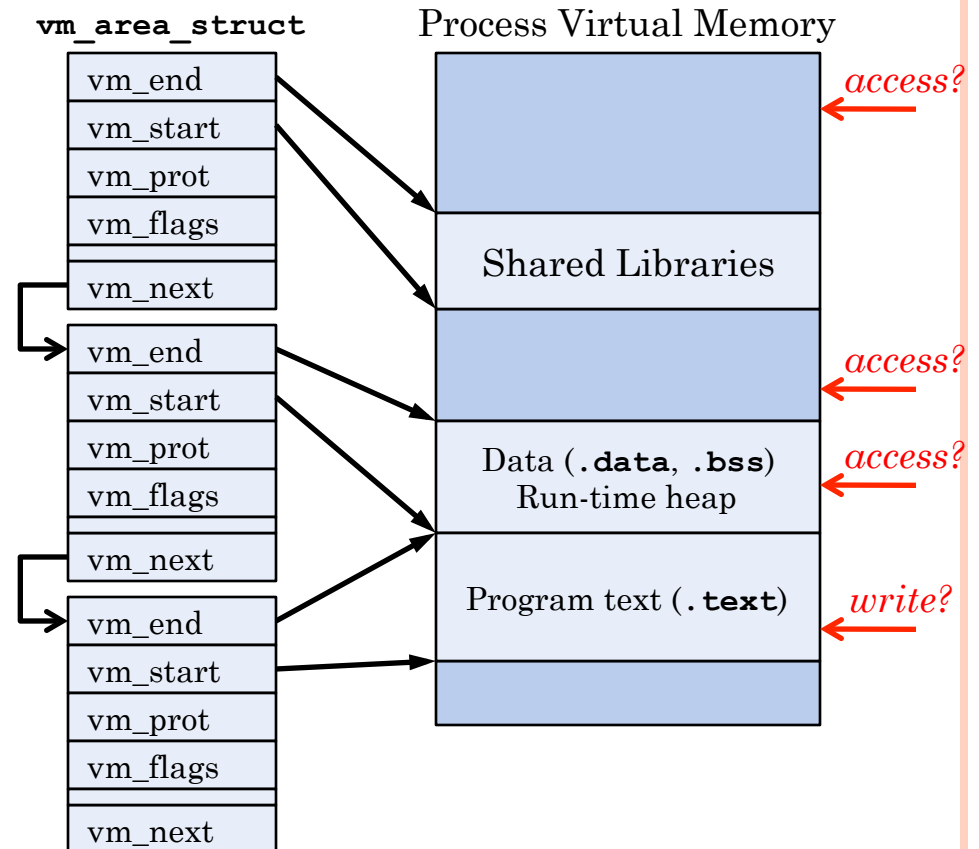
PROCESS VIRTUAL MEMORY AREAS (2)

- **vm_area_struct** fields specify details of each memory area
 - **vm_start**, **vm_end** specify extent of the memory area
 - **vm_prot** specifies read/write permissions for the memory area
 - **vm_flags** specifies whether memory area is shared among processes, or private to this process
- Normal memory accesses:
 - (Page is in memory, and the operation is allowed)
 - No intervention needed from the kernel...
 - CPU and MMU handle these accesses without any trouble



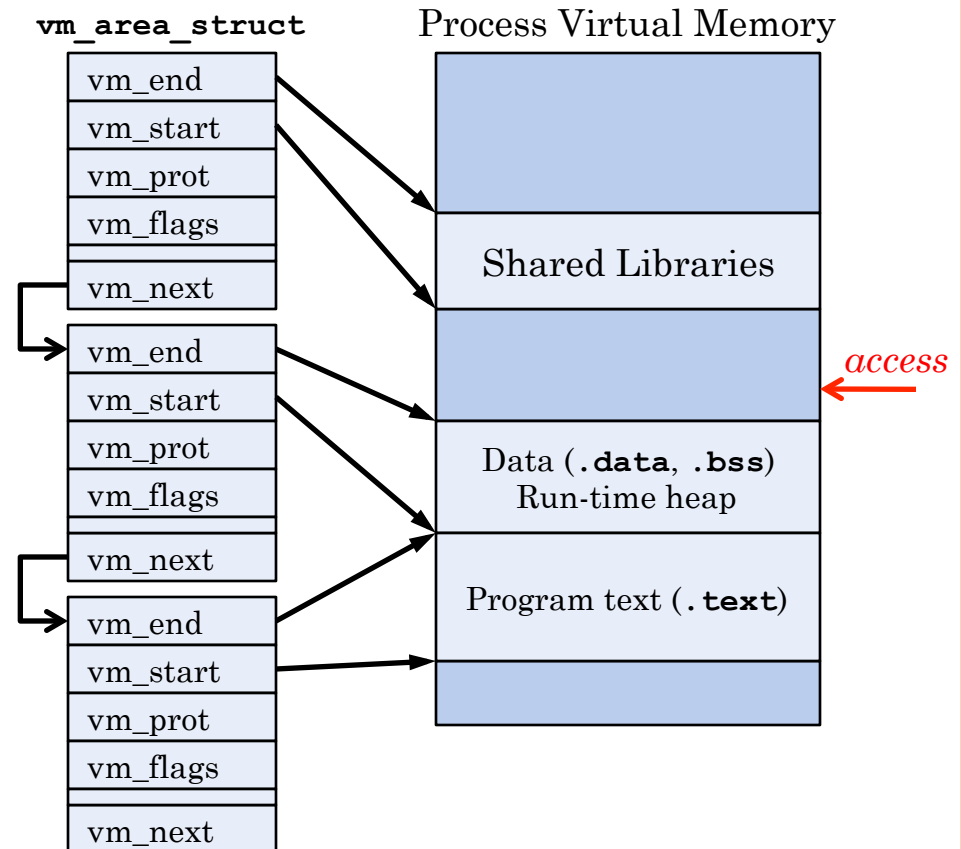
FAULTS!

- When a page fault or general protection fault occurs, the kernel must handle the situation!
- Faults can occur for many different reasons...
- Invalid accesses:
 - Program tried to write read-only memory
 - Program tried to access kernel-only memory
- Valid accesses:
 - Accessed page is in the swap device, not DRAM
 - Accessed page hasn't been allocated to program
- Kernel must decide how to handle each fault



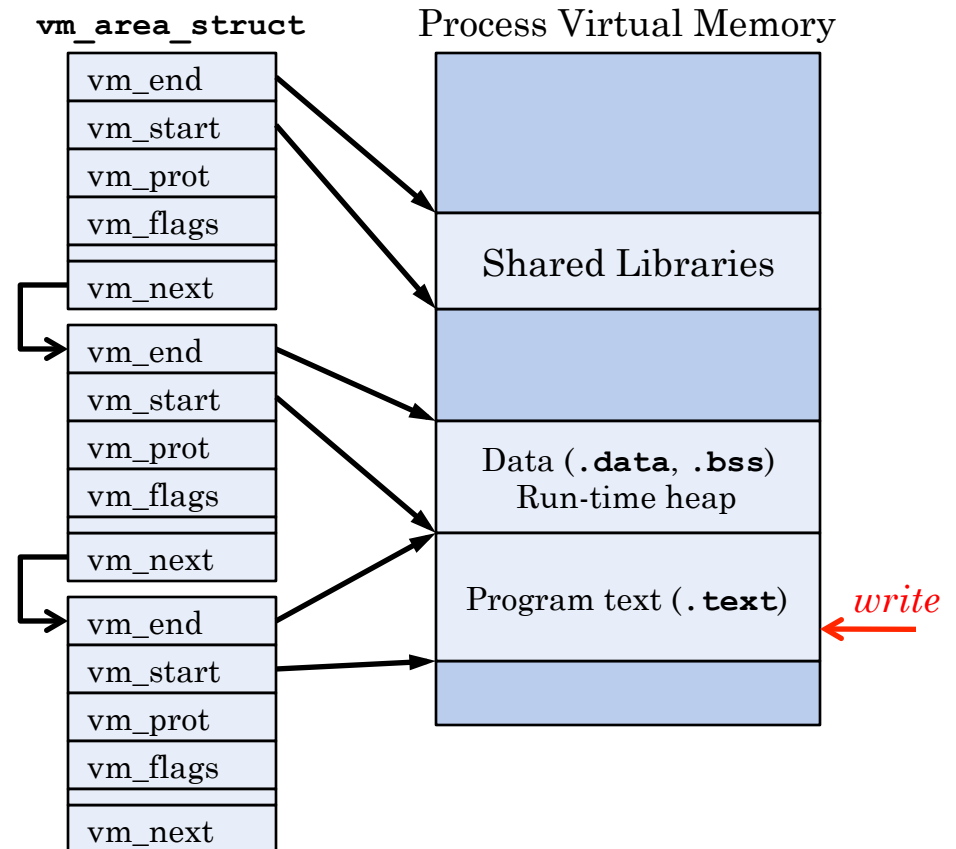
FAULTS! (2)

- The process' **vm_area_struct**-list tells the kernel how each fault should be handled
- If the program accesses a non-existent page:
 - MMU raises a page fault
- Kernel must check all area structs to see if the address itself is valid
 - Does it fall within some **vm_start** and **vm_end**?
- If not a valid address, a segmentation-fault signal is sent to the process
 - Default handler: terminate the process!



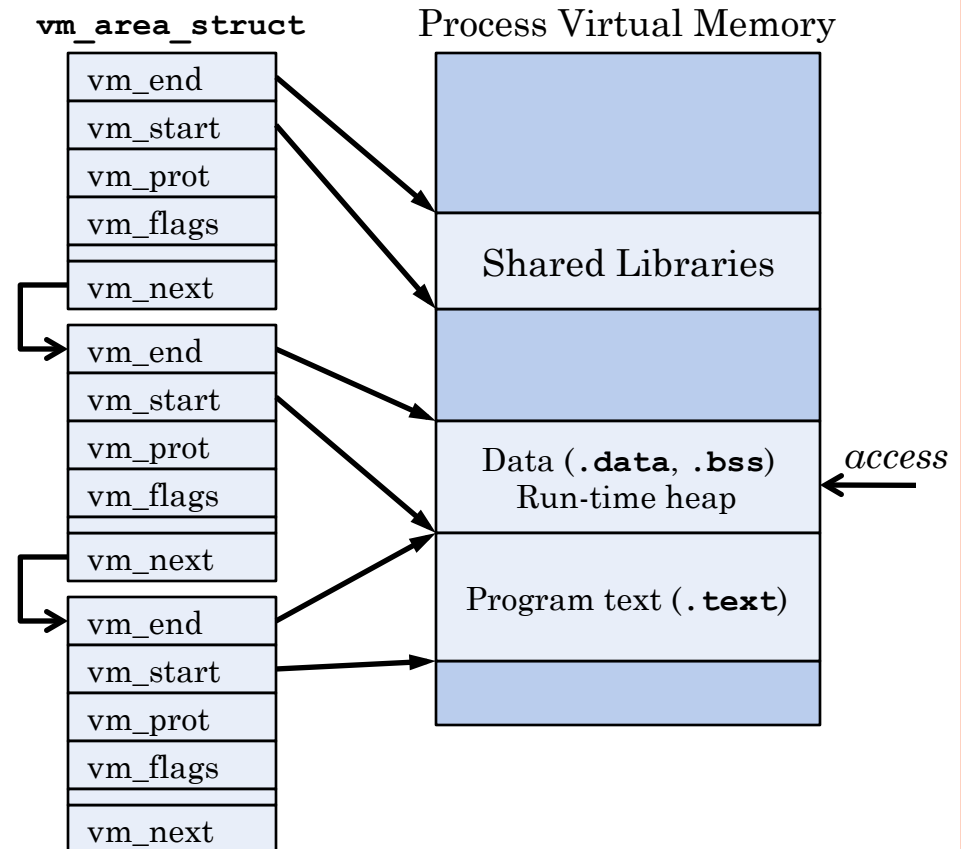
FAULTS! (3)

- If kernel determines that the address is valid, it must next check if the operation is valid
 - Is the process writing to read-only memory?
 - Is the process accessing kernel-only memory?
 - MMU raises a general protection fault
- If operation is invalid, a protection signal is delivered
 - Again, the process gets terminated.



FAULTS! (4)

- At this point, the kernel knows that the address is valid, and the operation is allowed
- Perform normal page-load operations:
 - Select victim page to evict
 - If victim page is dirty, write it back to disk
 - Load requested virtual page into memory
 - Return from fault handler
- CPU restarts instruction that caused the fault
 - This time, the instruction succeeds, since page is now in main memory



NEXT TIME

- Covered most of how the kernel can provide a useful virtual memory abstraction...
- Next time, finish up with a few higher-level abstractions that operating systems build on top of virtual memory