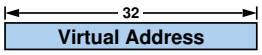
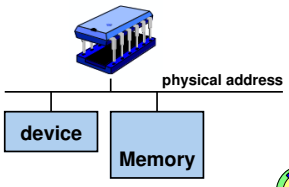
Housekeeping (Lecture 18 - 3/9,10/2016)

- Kernel 1 due at 11:45pm this Friday, 3/11/2016
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
- *Grading guidelines* is the only way we will grade
  - when running `faber_thread_test()`, you need to make sure that all the *exit codes* are correct
    - read the code to figure out what values to expect
  - you should be able to run commands after commands, etc.
  - if you are confused about "SELF-checks", please send me e-mail
- After submission, make sure you *Verify Your Kernel Submission*
  - tests in sections (C), (D), and (E) of the grading guidelines must run in the "foreground"
- This Friday, the TAs will give an introduction to Kernel 2
- By the way, *midterm* exam does cover *kernel 1*

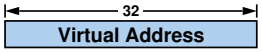
Virtual Address



- Who uses *virtual address*?
  - user processes
  - kernel processes
  - pretty much every piece of software
- You would use a virtual address to address any memory location in the 32-bit address space
- Anything uses *physical address*?
  - nothing in OS
  - well, the hardware uses physical address (and the processor is hardware)
  - the OS *manages* the *physical address space*

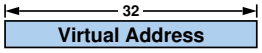


Virtual Address



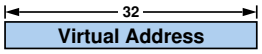
- To *access a memory location*, you need to specify a *memory address*
  - in a user process (or even a kernel process), you would use a *virtual address* to address any memory location in the 32-bit address space
- Why would you want to access a memory location?
  - e.g., to fetch a machine instruction
    - you need to specify a memory location to fetch from
    - how do you know which memory location to fetch from?
      - EIP (on an x86 machine), which contains a virtual address

Virtual Address



- To *access a memory location*, you need to specify a *memory address*
  - in a user process (or even a kernel process), you would use a *virtual address* to address any memory location in the 32-bit address space
- Why would you want to access a memory location?
  - e.g., to fetch a machine instruction
  - e.g., to push EBP onto the stack
    - you need to specify a memory location to store the content of EBP
    - how do you know which memory location to write to?
      - ESP, which contains a virtual address

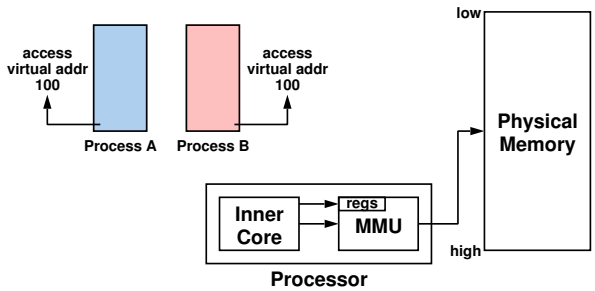
Virtual Address



- To *access a memory location*, you need to specify a *memory address*
  - in a user process (or even a kernel process), you would use a *virtual address* to address any memory location in the 32-bit address space
- Why would you want to access a memory location?
  - e.g., to fetch a machine instruction
  - e.g., to push EBP onto the stack
  - e.g., `x = 123`, where `x` is a local variable
    - you need to specify a memory location to write `123` to
    - how do you know which memory location to wrote to?
      - EBP, which contains a virtual address

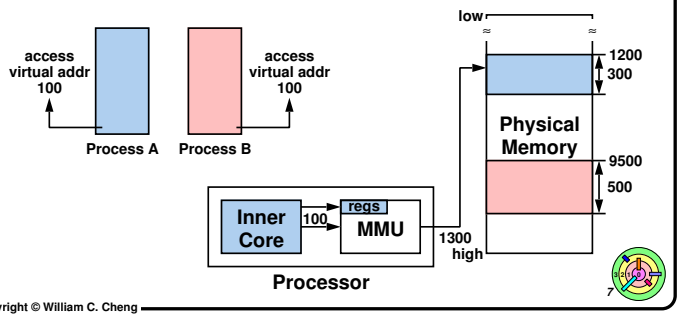
Basic Idea: Address Translation

- One level of *indirection* with a *Memory Management Unit (MMU)*
  - don't address physical memory directly
    - address out of CPU inner core is *virtual*
  - use a *Memory Management Unit (MMU)*
    - virtual address is *translated* into physical address via MMU



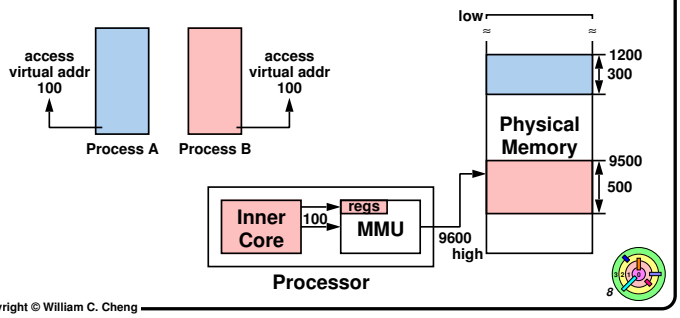
Basic Idea: Address Translation

- One level of *indirection* with a *Memory Management Unit (MMU)*
  - don't address physical memory directly
    - address out of CPU inner core is *virtual*
  - use a *Memory Management Unit (MMU)*
    - virtual address is *translated* into physical address via MMU



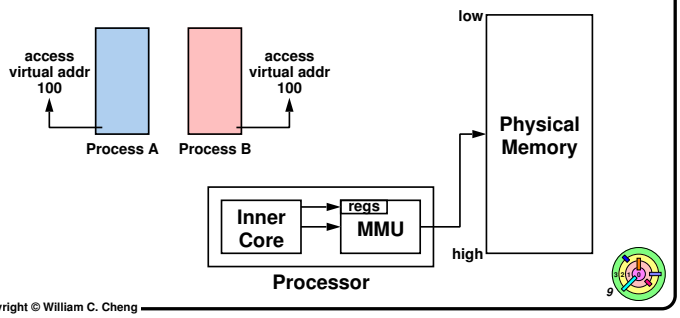
Basic Idea: Address Translation

- One level of *indirection* with a *Memory Management Unit (MMU)*
  - don't address physical memory directly
    - address out of CPU inner core is *virtual*
  - use a *Memory Management Unit (MMU)*
    - virtual address is *translated* into physical address via MMU

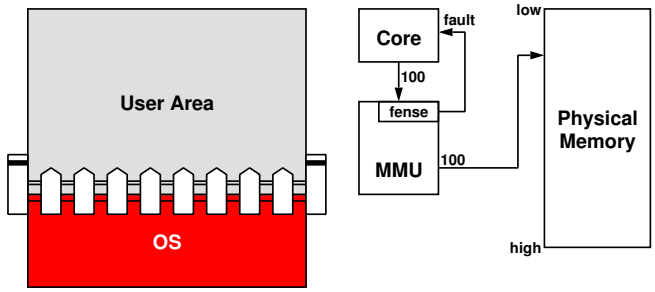


Address Translation

- Protection/isolation
- Illusion of large memory
- Sharing
- New abstraction (such as memory-mapped files)

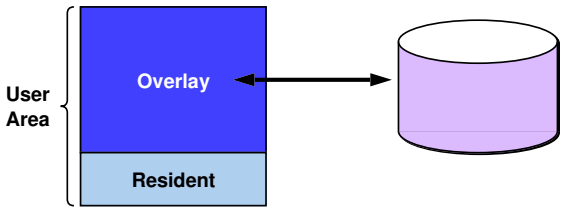


Memory Fence



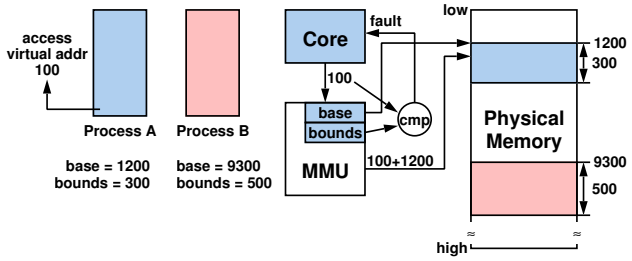
- In the old days
  - if a user program tries to access OS area, *hardware* (very simple MMU) will generate a trap
  - does not protect user pocesses from each other
    - there's only one user process anyway

Memory Fence and Overlays



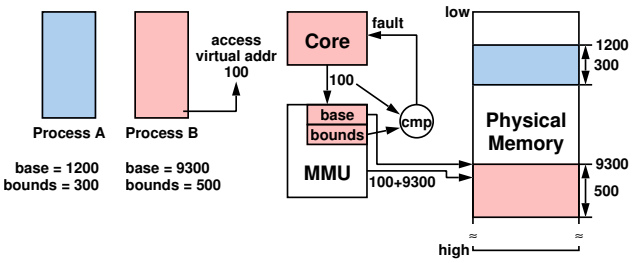
- What if the user program won't fit in memory?
  - use *overlays*
  - programmers (not the OS) have to keep track of which overlay is in physical memory and deal with the complexities of managing *overlays*

Base and Bounds Registers



- Multiple user processes
  - OS maintains a pair of registers for each user process
    - *bounds register:* address space size of the user process
    - *base register:* start of physical memory for the user process
  - addresses *relative* to the *base register*
  - memory reference >= 0 and < bounds, *independent* of *base* (this is known as "position independence")

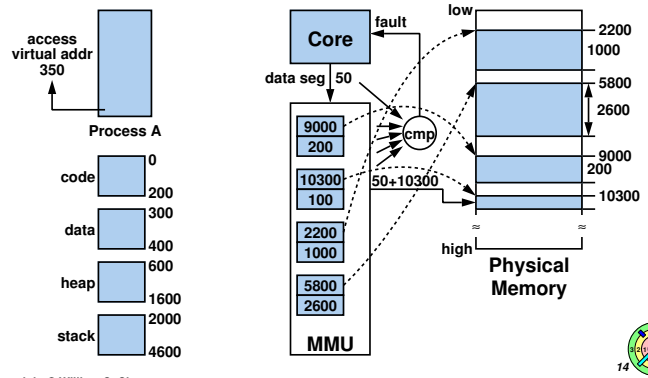
Base and Bounds Registers



- Multiple user processes
  - OS maintains a pair of registers for each user process
    - *bounds register:* address space size of the user process
    - *base register:* start of physical memory for the user process
  - addresses *relative* to the *base register*
  - memory reference >= 0 and < bounds, *independent* of *base* (this is known as "position independence")

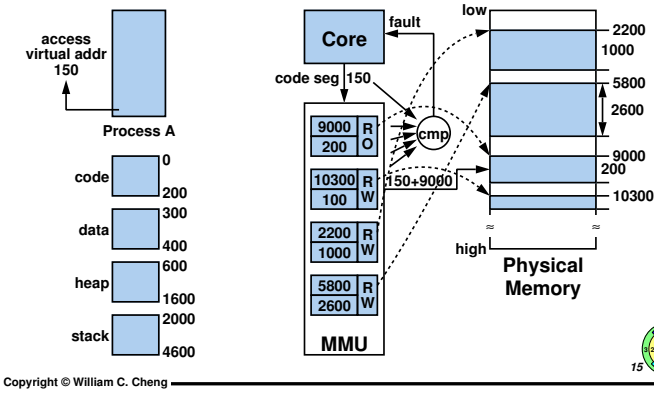
Generalization of Base and Bounds: Segmentation

- One pair of *base* and *bounds* registers *per segment*
  - code, data, heap, stack, and may be more
  - compiler compiles programs into segments



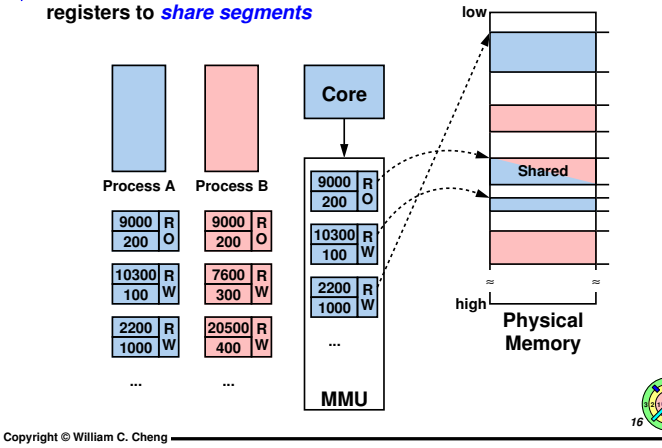
Access Control With Segmentation

- Access control / protection
  - *read-only*, *read/write*



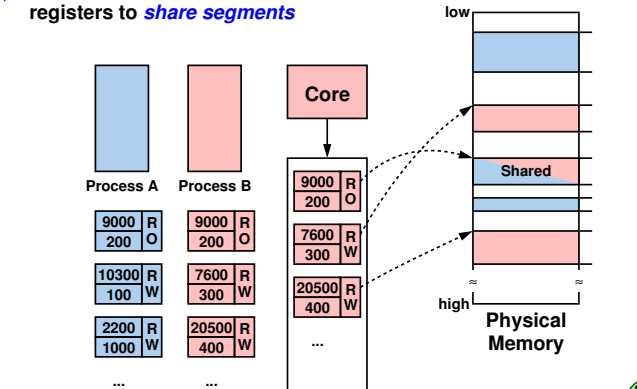
Sharing Segments

- Can simply setup base and bounds registers to *share segments*



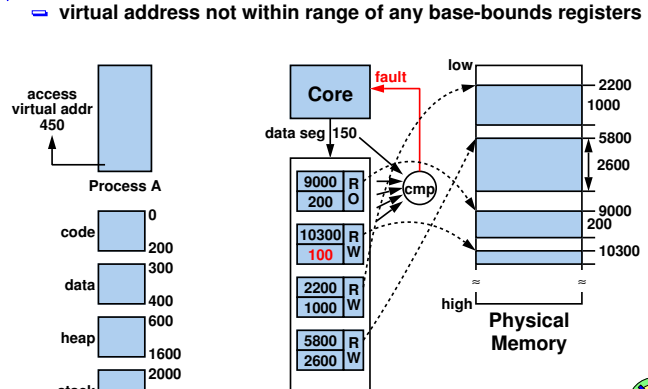
Sharing Segments

- Can simply setup base and bounds registers to *share segments*



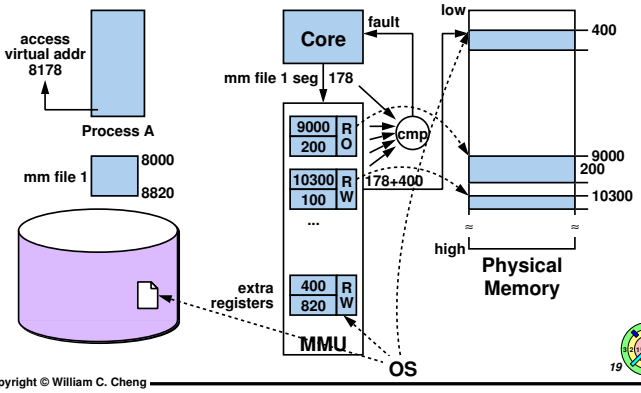
Segmentation Fault

- *Segmentation fault*
  - virtual address not within range of any base-bounds registers



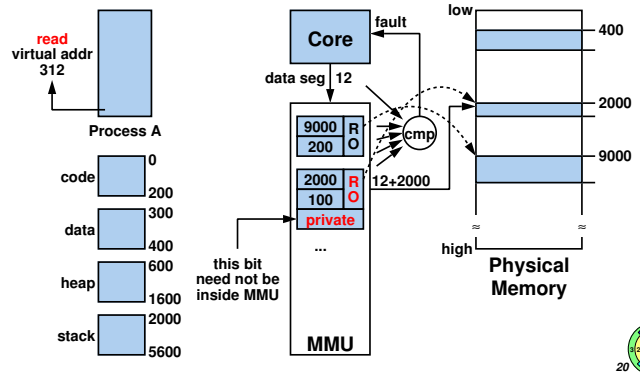
Memory Mapped File

- *Memory Mapped File*
  - the `mmap()` system call
  - can map an entire file (or part of it) into a segment



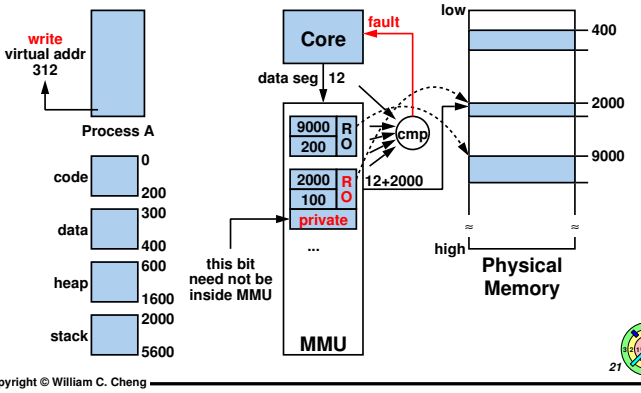
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



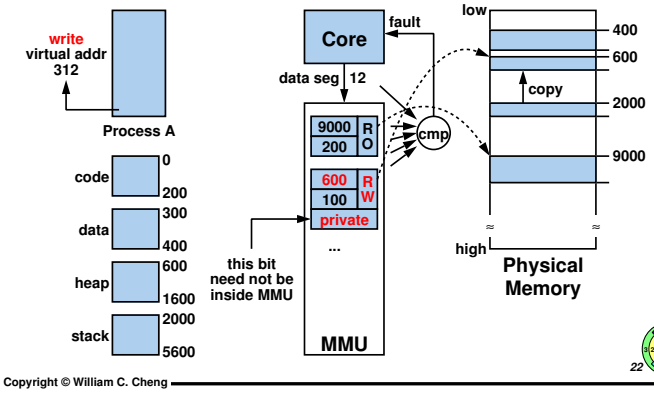
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



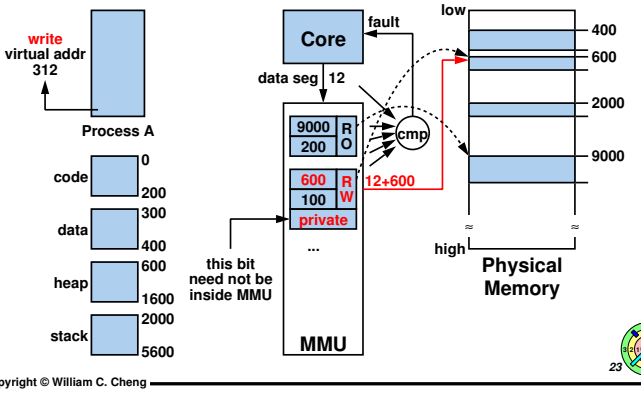
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



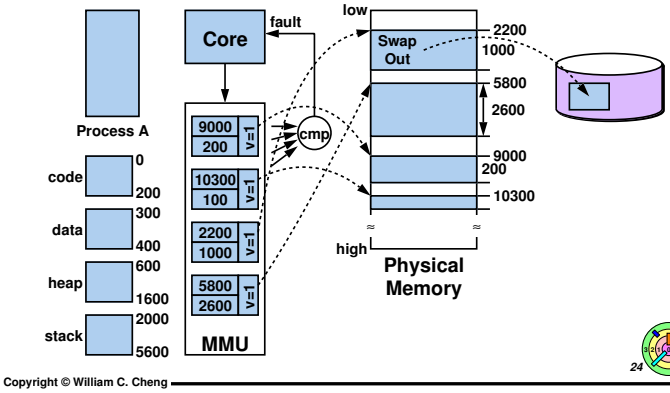
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



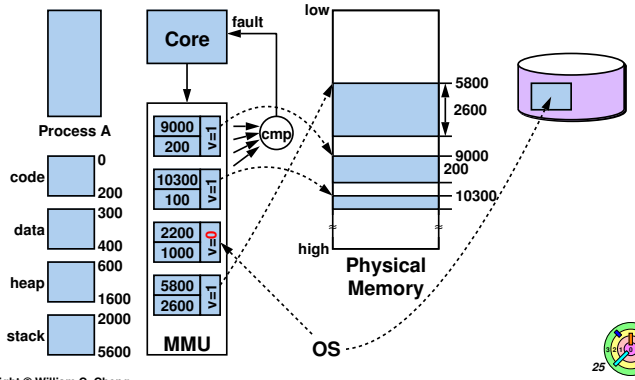
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



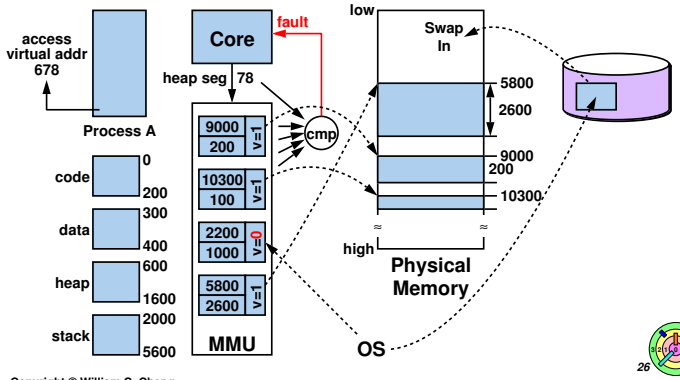
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



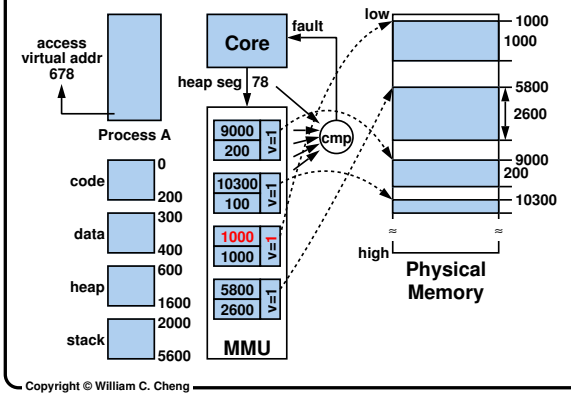
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



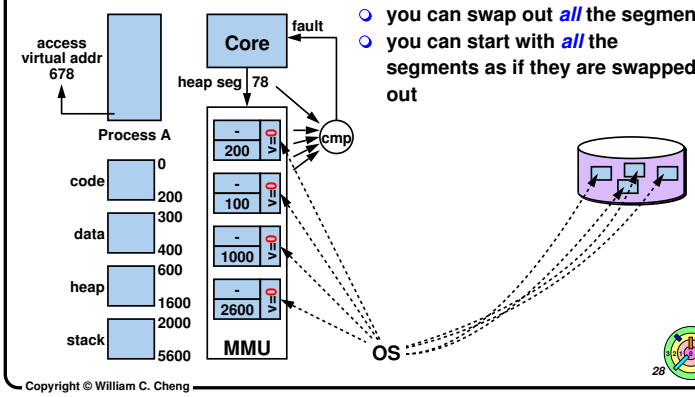
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)

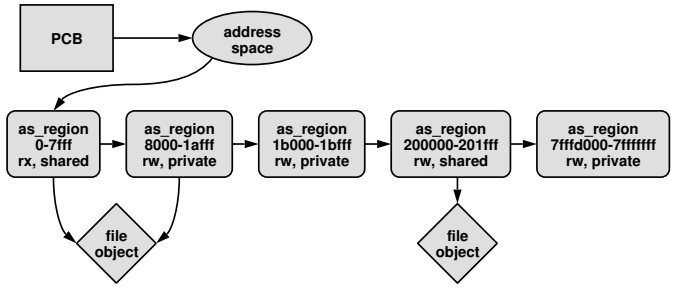


Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)
    - you can swap out *all* the segments
    - you can start with *all* the segments as if they are swapped out

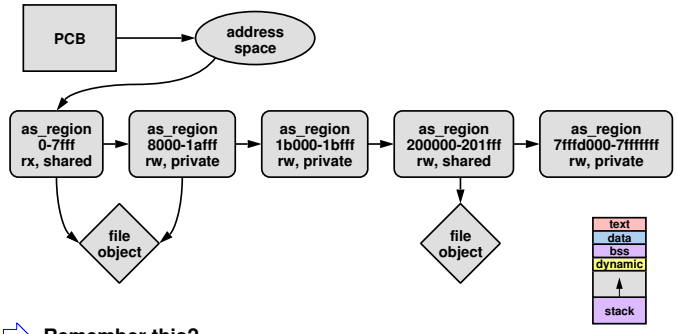


Swapping / Backing Store



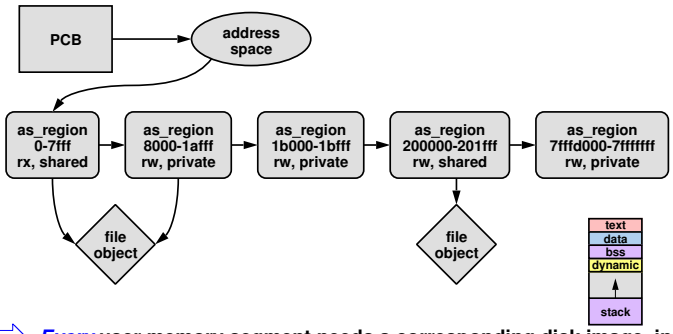
- Remember this?

Swapping / Backing Store



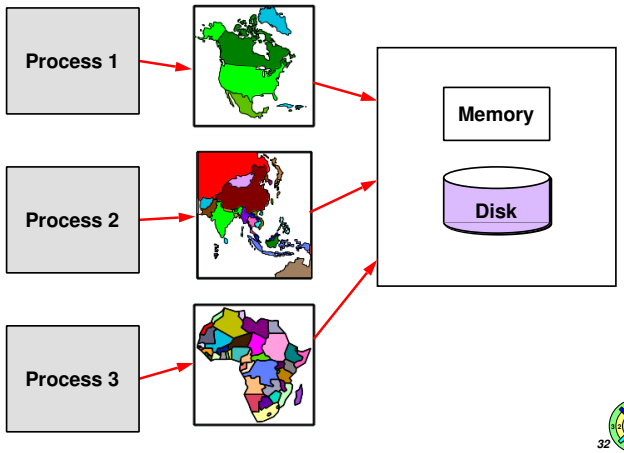
- Remember this?
  - this is the representation of the *address space* of a user process
    - each segment corresponds to an `as_region`

Swapping / Backing Store



- *Every* user memory segment needs a corresponding disk image, in case the user process needs to be *swapped out*
  - some kernel memory can be *locked down* to prevent it from being swapped out accidentally

Virtual Memory



7.2 Hardware Support for Virtual Memory

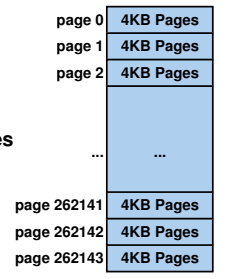
- *Forward-Mapped Page Tables*
- Linear Page Tables
- Hashes Page Tables
- Translation Lookaside Buffers
- 64-Bit Issues
- Virtualization

Structuring Virtual Memory

- *Segmentation* (just discussed)
  - divide the address space into variable-size segments (typically each corresponding to some logical unit of the program, such as a module or subroutine)
  - external fragmentation possible
  - "first-fit" is slow
  - not very common these days
- *Paging*
  - divide the address space into fixed-size pages
  - internal fragmentation possible

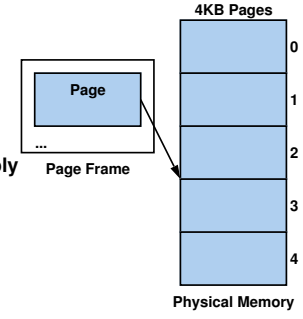
Paging

- Map *fixed-size pages* into physical memory *(into physical pages)*
  - address space is divided into pages
    - indexed by virtual page number
  - physical memory is divided into pages (of the same size)
    - indexed by physical page number
  - need a loopup table to map *virtual page numbers* to *physical page numbers*
- Ex: 1GB of physical memory with 4KB pages
  - 2^{18} physical pages
  - an address (either physical or virtual) is *page-aligned* if its least significant 12 bits are all zero
- Many *hardware* mapping techniques
  - *MMU* and *page table* (mostly in software)
  - *translation lookaside buffers (TLB)*



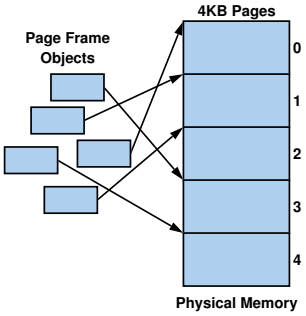
Page Frames

- A *page frame* data structure / object is used to maintain information about physical pages and their association with important kernel data structures
  - contains a *physical page number*
  - there is a *one-to-one mapping* between page frames and physical pages
    - we use "page frame" and "physical page" interchangeably

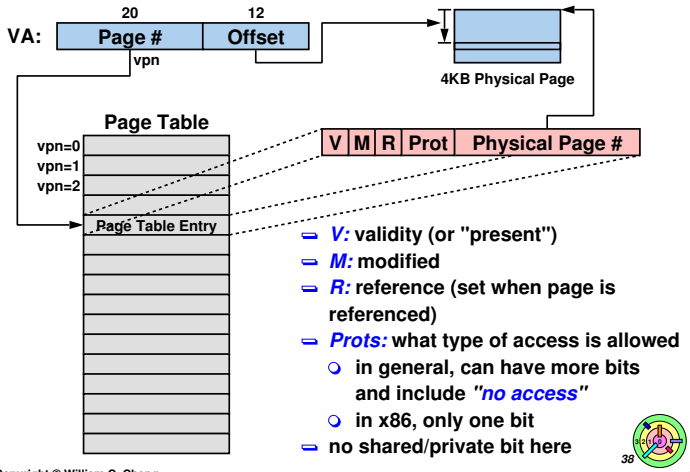


Page Frames

- It is important to be able to perform both *forward lookup* and *reverse lookup*
  - given a virtual address of a process, find page frame
  - given a page frame, find processes and virtual addresses that uses this page frame
  - `weenix` page frame data structure is a bit involved
    - see kernel 3 FAQ
    - the kernel must use a *virtual address* to write into a physical page or read the content of a physical page

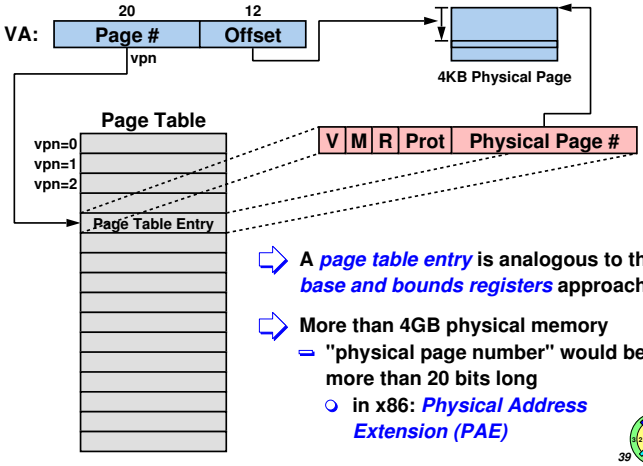


Basic (Two-level) Page Tables



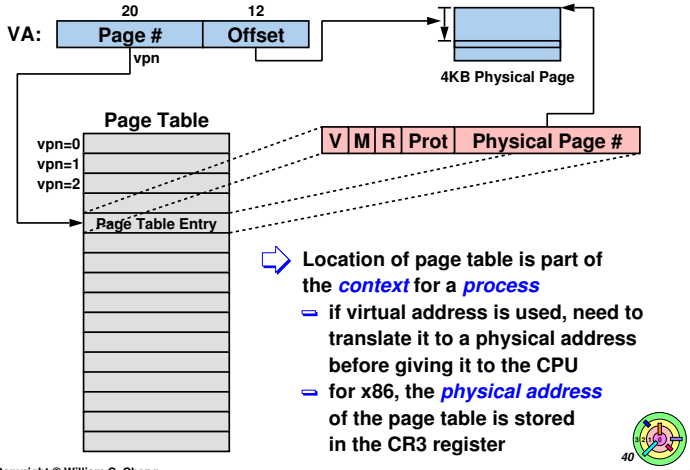
- *V:* validity (or "present")
- *M:* modified
- *R:* reference (set when page is referenced)
- *Prots:* what type of access is allowed
  - in general, can have more bits and include *"no access"*
  - in x86, only one bit
- no shared/private bit here

Basic (Two-level) Page Tables



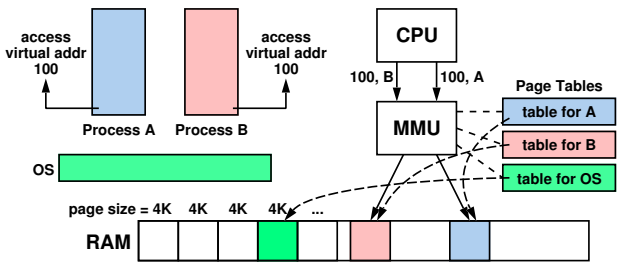
- A *page table entry* is analogous to the *base and bounds registers* approach
- More than 4GB physical memory
  - "physical page number" would be more than 20 bits long
    - in x86: *Physical Address Extension (PAE)*

Basic (Two-level) Page Tables



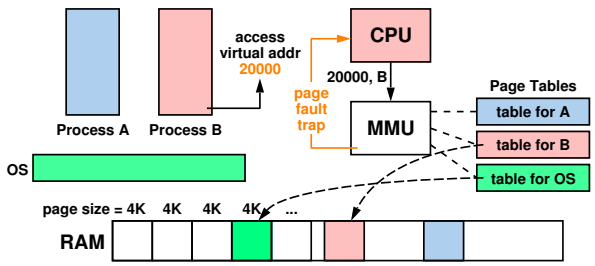
- Location of page table is part of the *context* for a *process*
  - if virtual address is used, need to translate it to a physical address before giving it to the CPU
  - for x86, the *physical address* of the page table is stored in the CR3 register

Page Table



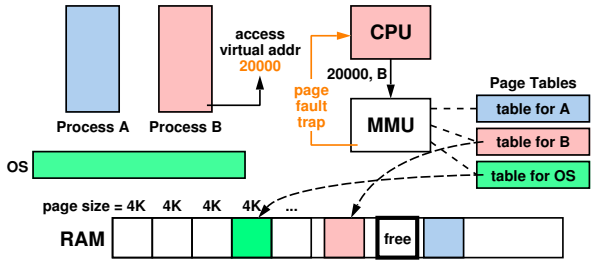
- A page table *(usually sits in physical memory)* is associated with each *process*
  - OS has its page table as well
- Memory Management Unit (MMU) maps virtual address to physical address
  - MMU got turned on some time during boot

Page Table

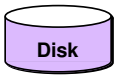


- *Page fault*
  - page table does not have the requested address

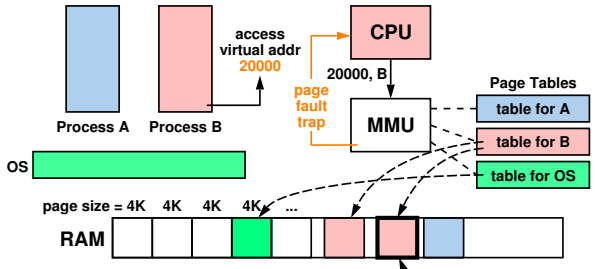
Page Table



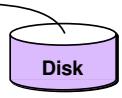
- ***Page fault***
  - page table does not have the requested address
  - OS finds a free page frame
    - what if no free page frame is available?



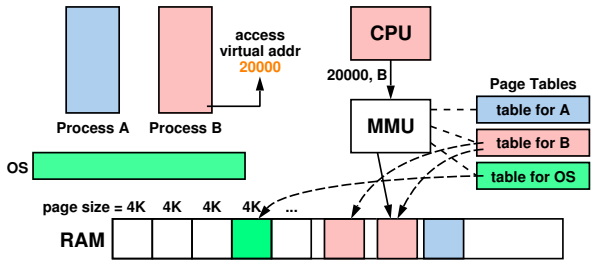
Page Table



- ***Page fault***
  - page table does not have the requested address
  - OS finds a free page frame
    - what if no free page frame is available?
  - OS loads the requested page from disk



Page Table



- ***Page fault***
  - page table does not have the requested address
  - OS finds a free page frame
    - what if no free page frame is available?
  - OS loads the requested page from disk
  - OS adjusts MMU and ***restarts*** user memory reference