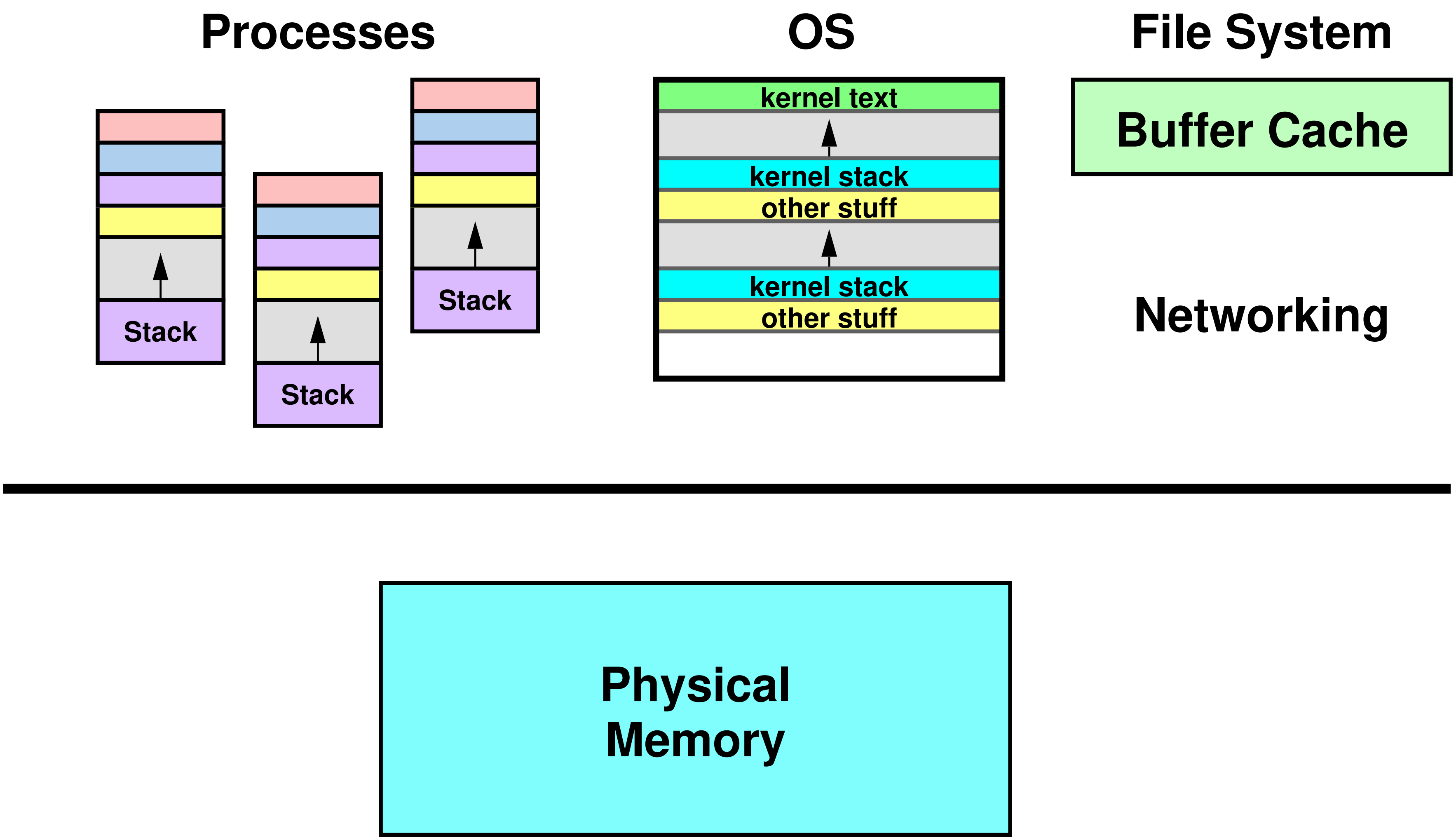
Ch 7: Memory Management

Bill Cheng

http://merlot.usc.edu/william/usc/

Memory Management

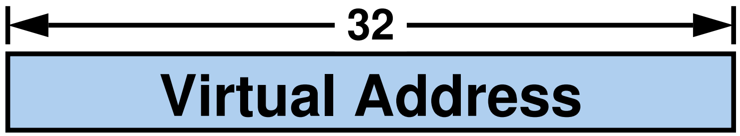


- Challenges
  - what to do when you run out of space?
  - protection

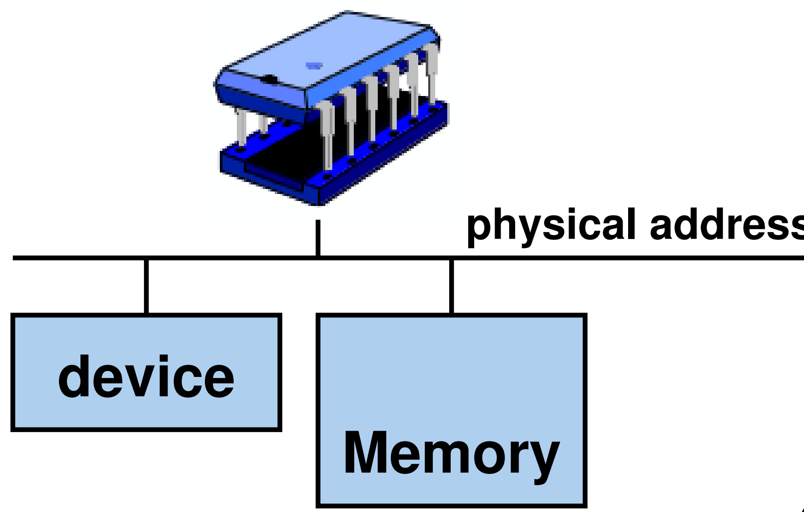
The Address-Space Concept

- Protect processes from one another
- Protect the OS from user processes
- Provide efficient management of available storage
  - illusion of large memory
  - sharing (code, data, communication)
  - new abstraction (such as pipes, memory-mapped files)

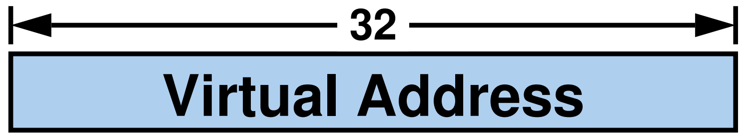
Virtual Address



- 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*



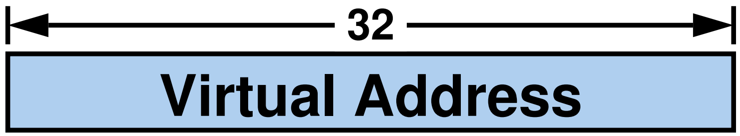
Virtual Address



- 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*

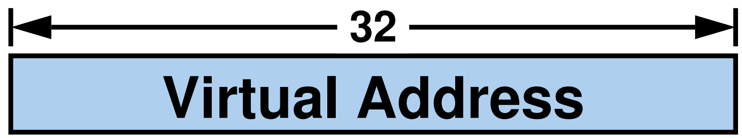
Virtual Address



- 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 EAX onto the stack
    - you need to specify a memory location to store the content of EAX
    - how do you know which memory location to write to?
      - ESP, which contains a *virtual address*

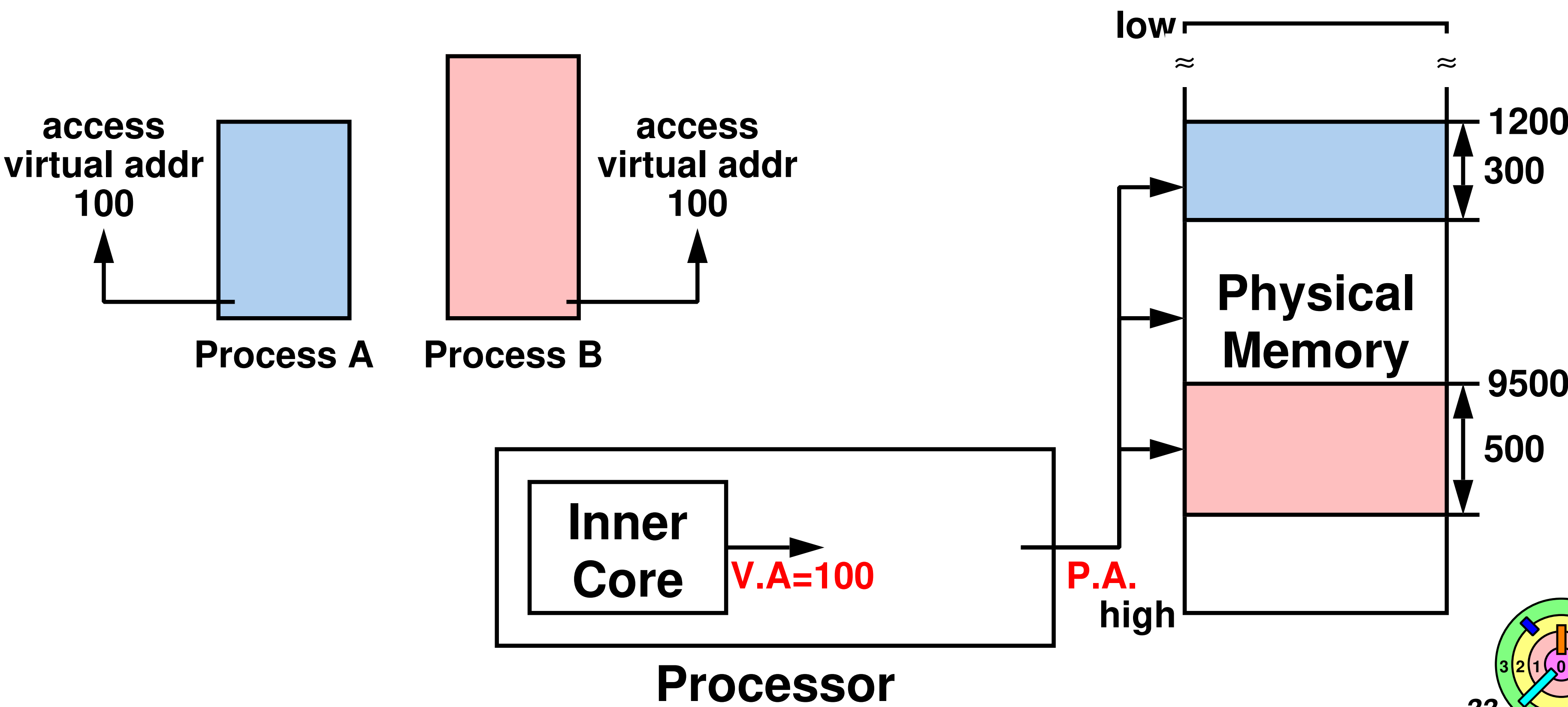
Virtual Address



- 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*
- Is there any CPU register that contains a physical address?

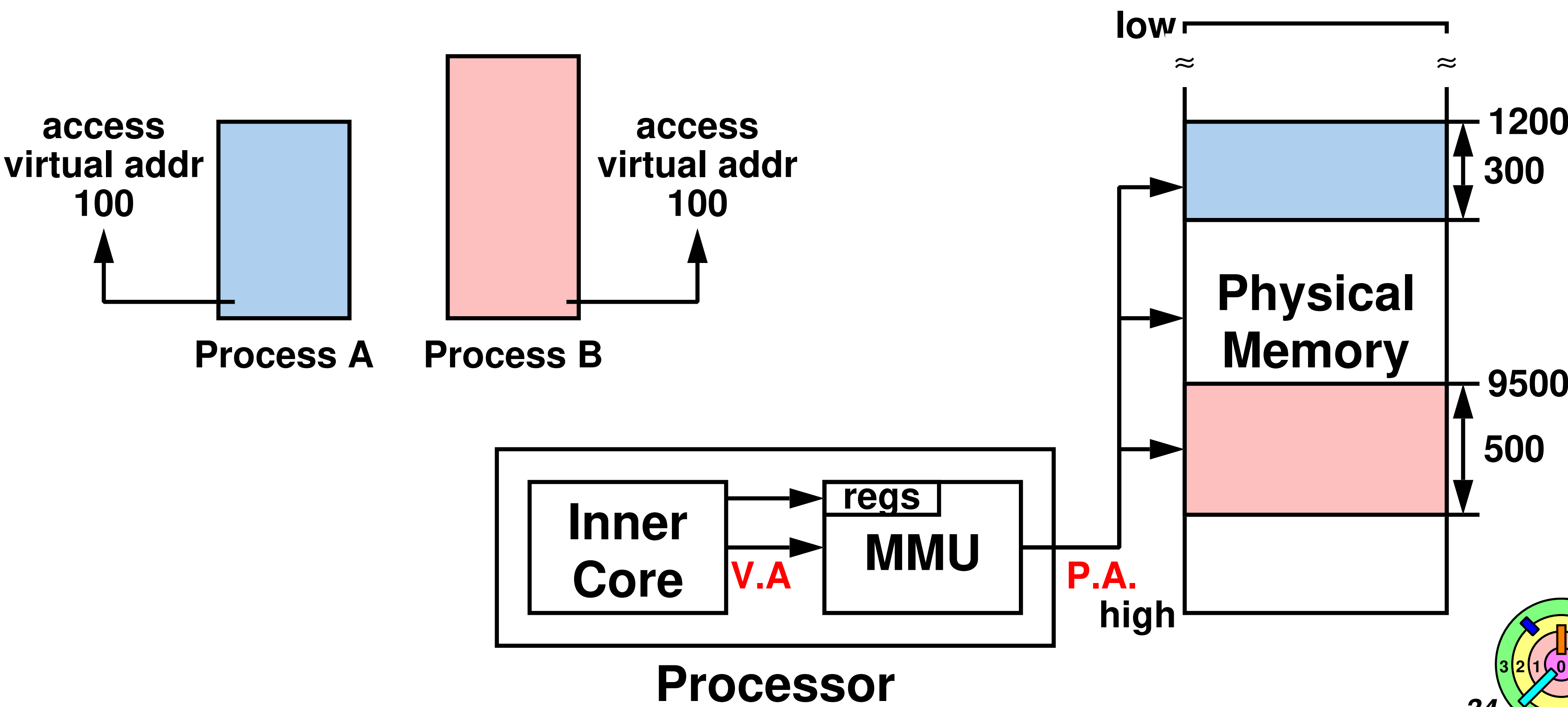
Basic Idea: Address Translation

- We want the same virtual address to get "translated" to a different physical address, depending on which process is running
  - how?



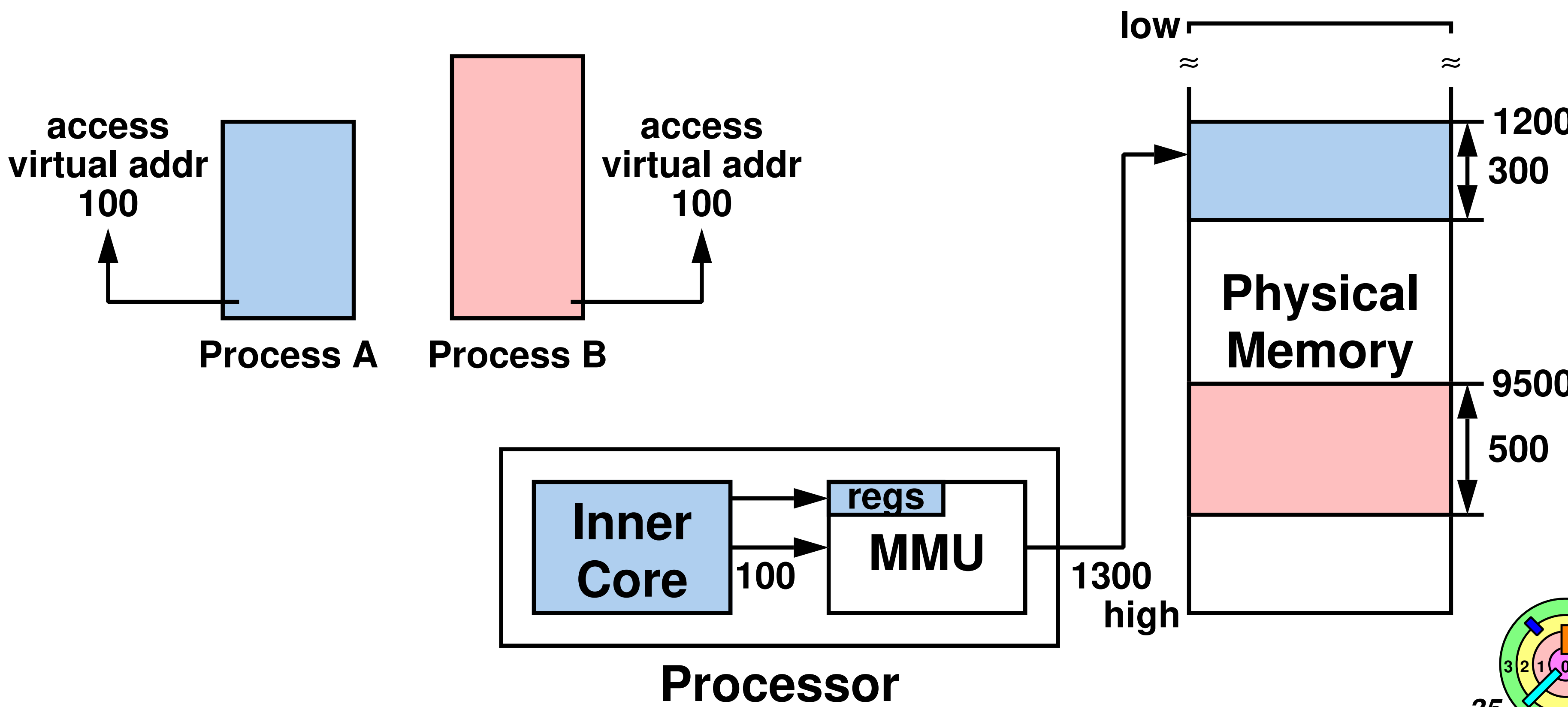
Basic Idea: Address Translation

- 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
    - physical memory can be located anywhere



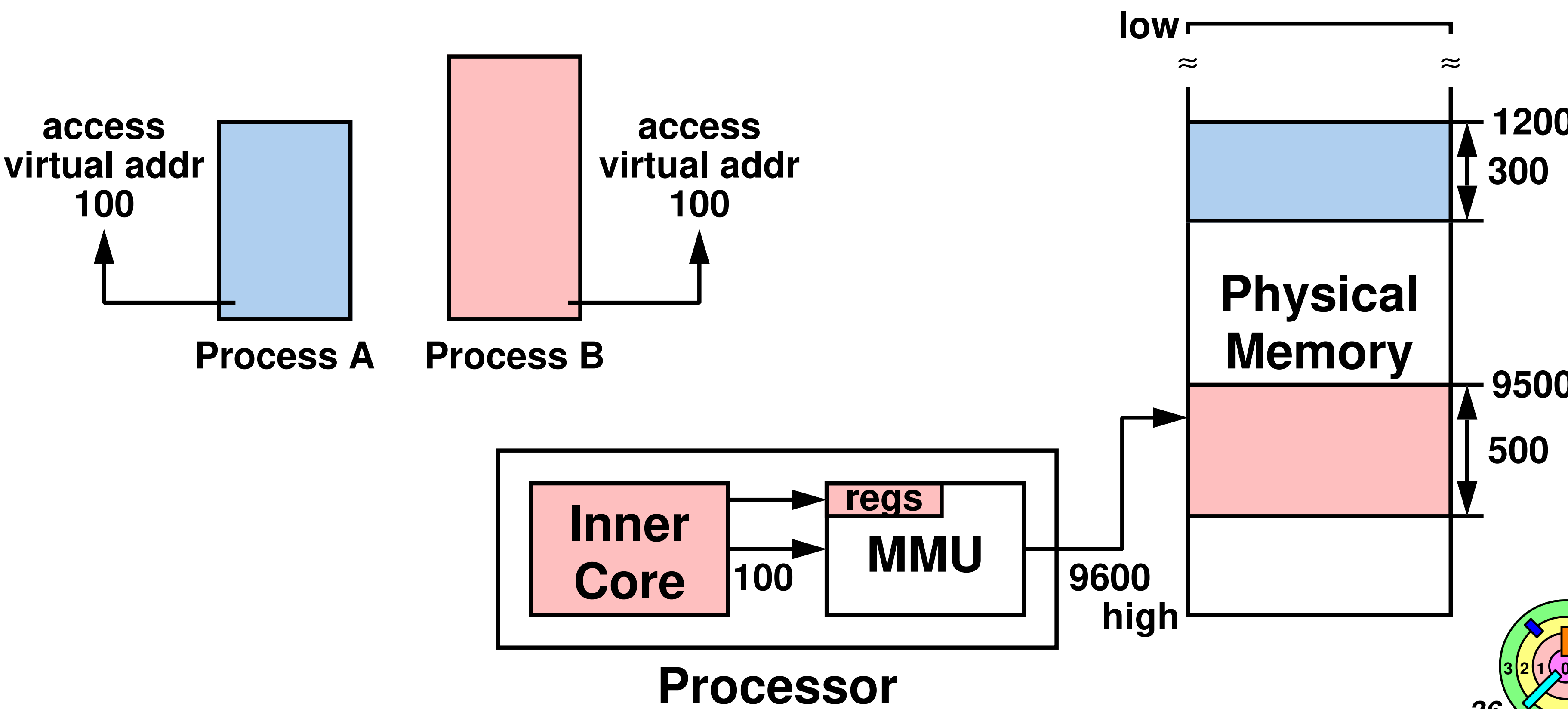
Basic Idea: Address Translation

- 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
    - physical memory can be located anywhere



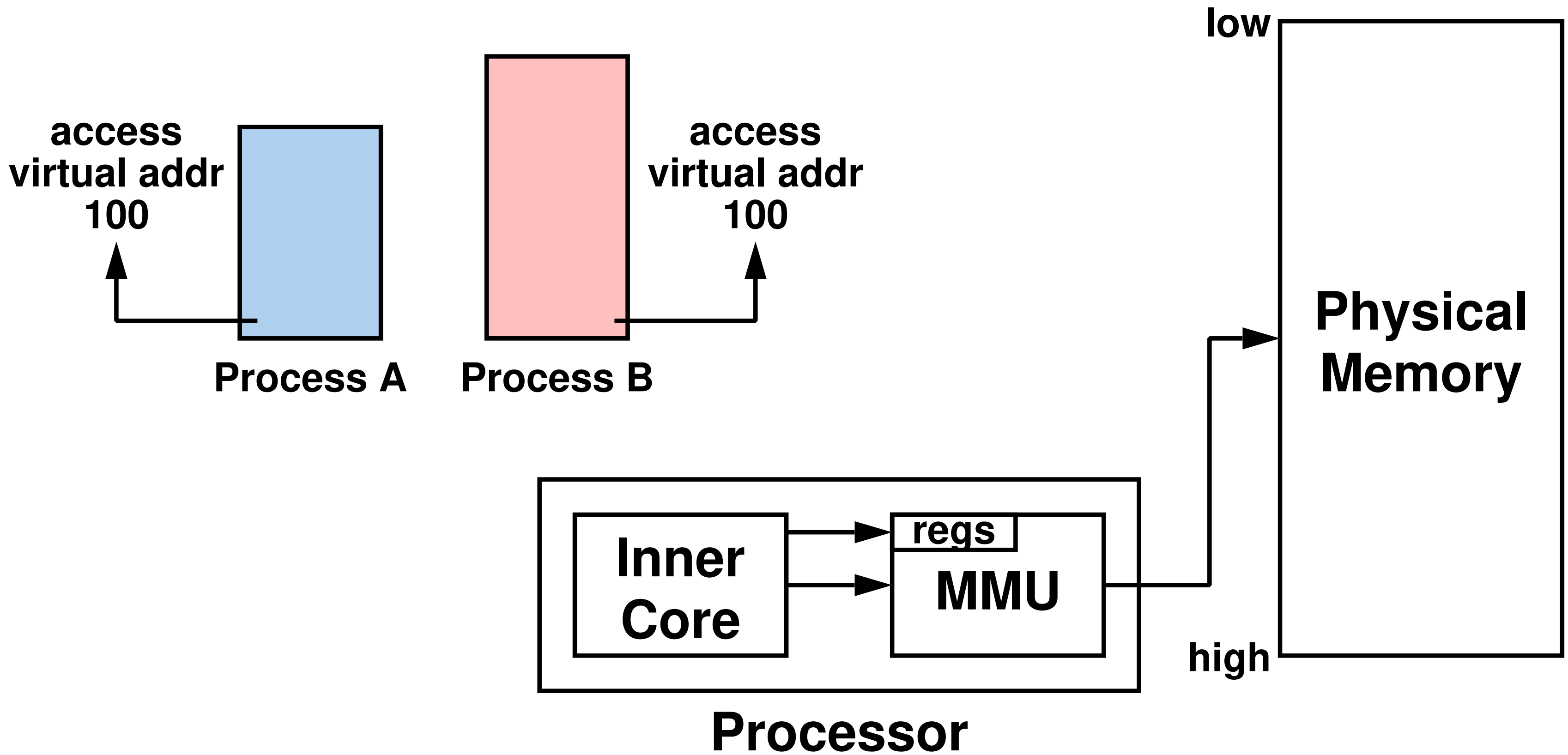
Basic Idea: Address Translation

- 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
    - physical memory can be located anywhere

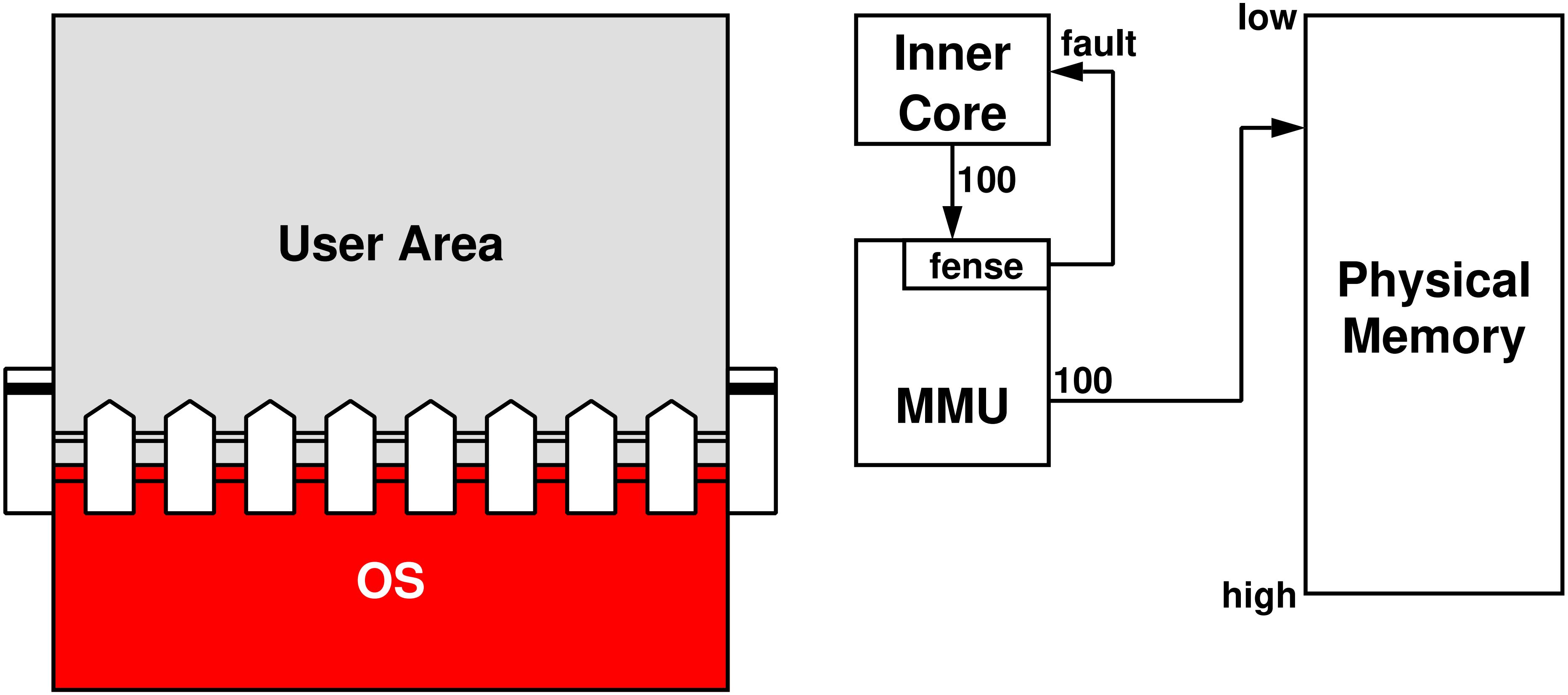


Address Translation

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

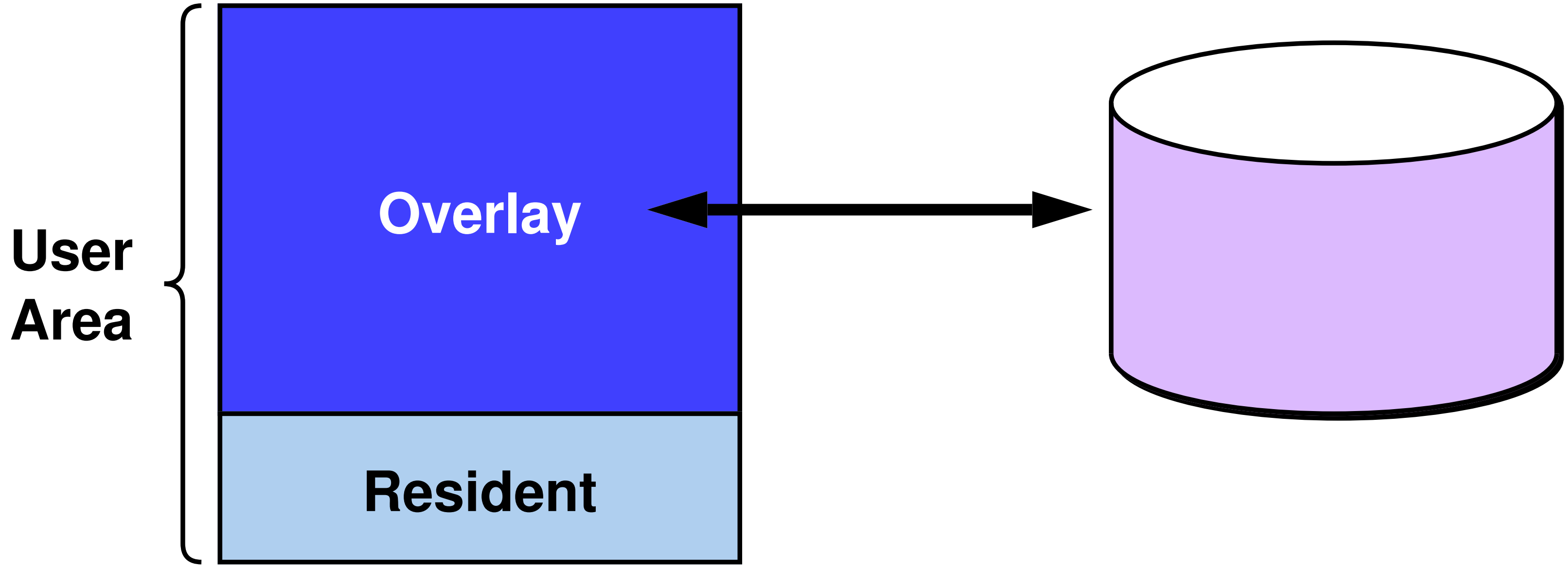


Memory Fence



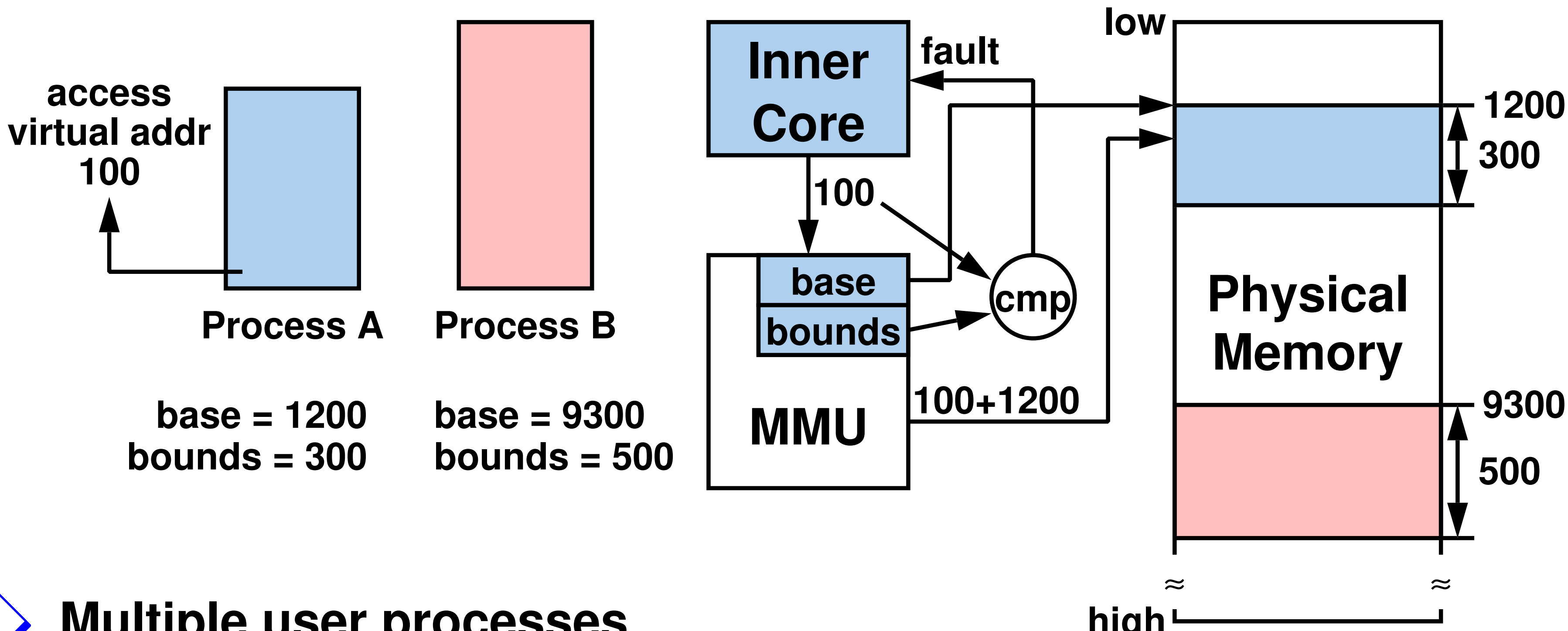
- In the old days
  - if a user program tries to access OS area, ***hardware*** (very simple MMU) will generate a trap
  - does not protect user pocesses from each other
    - there's only ***one user process*** anyway

Memory Fence and Overlays



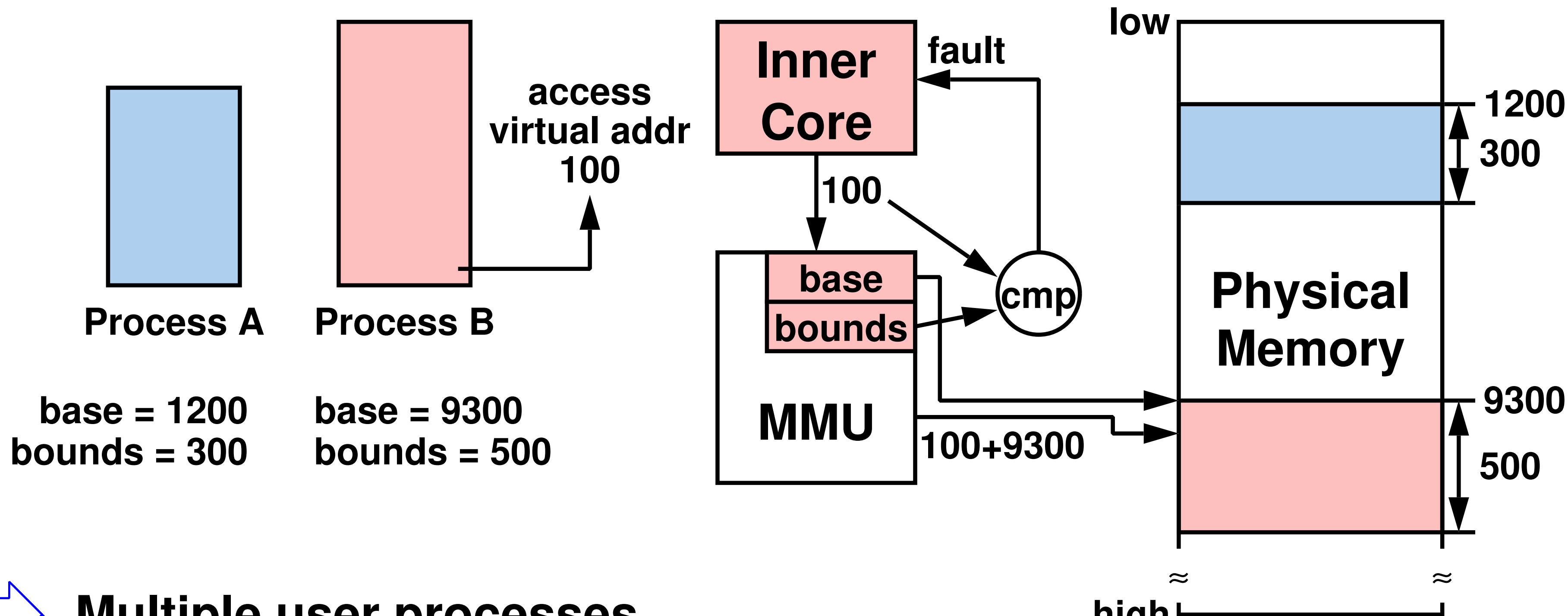
- What if the user program won't fit in memory?
  - use *overlays*
  - programmers (not the OS) have to keep track of which overlay is in physical memory and deal with the complexities of managing *overlays*

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
  - address *relative* to the *base register*
  - virtual memory reference >= 0 and < bounds, *independent* of *base* (this is known as "*position independence*")

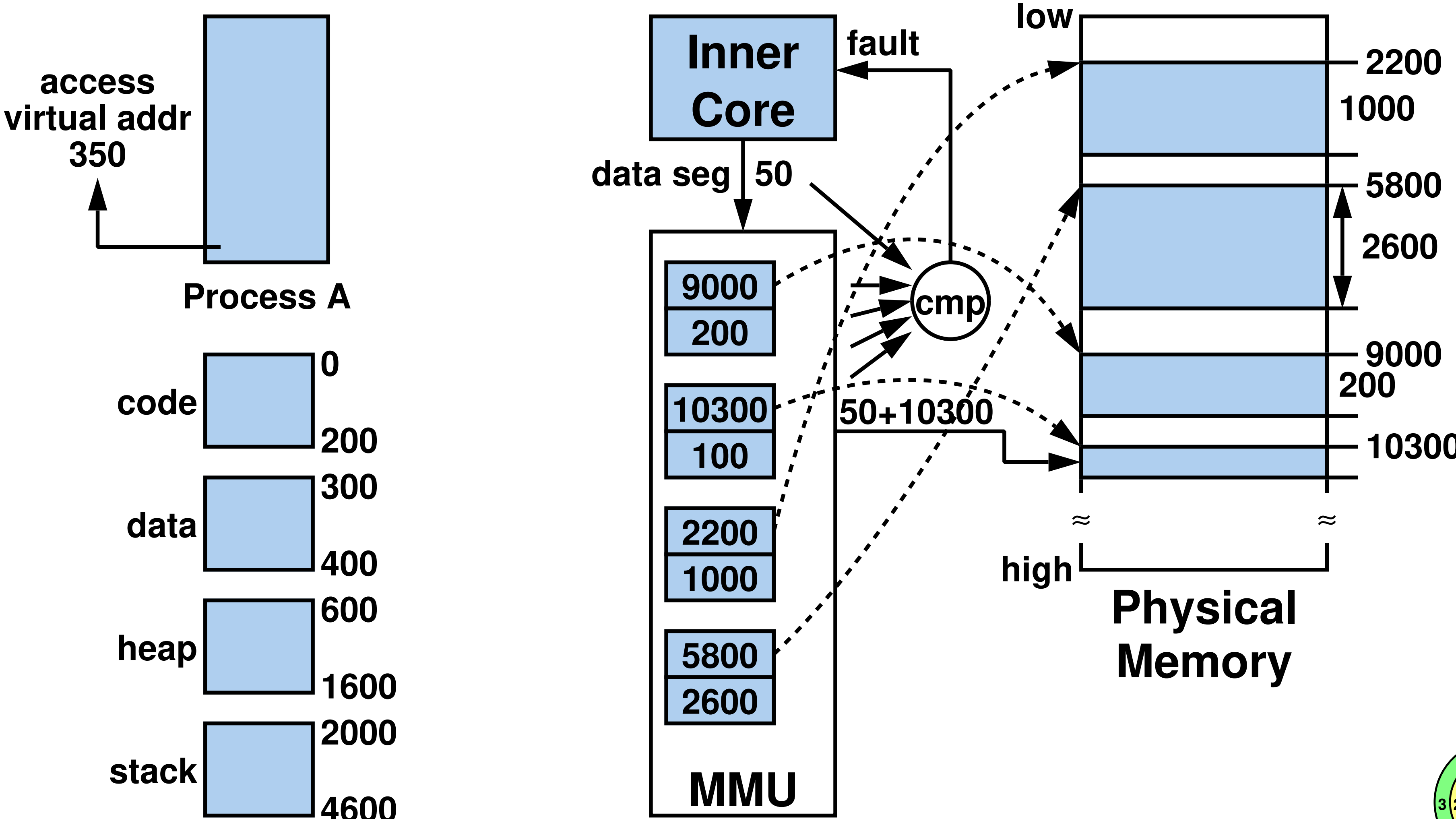
Base and Bounds Registers



- 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
- MMU registers are part of the *context* of a process
  - in kernel 1, a PCB has something called `pagedir` (MMU register for x86 CPU uses a different scheme)

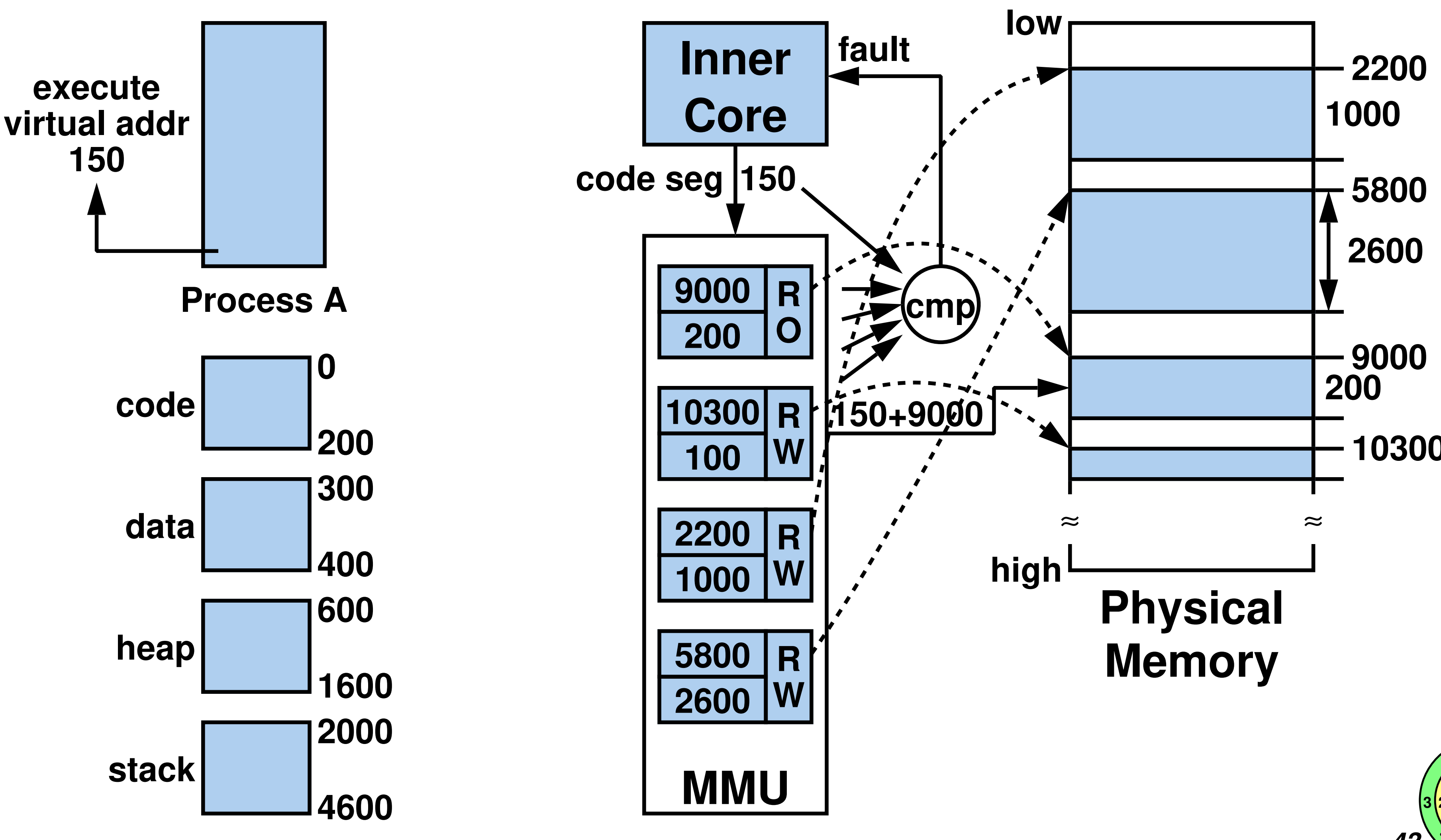
Generalization of Base and Bounds: Segmentation

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



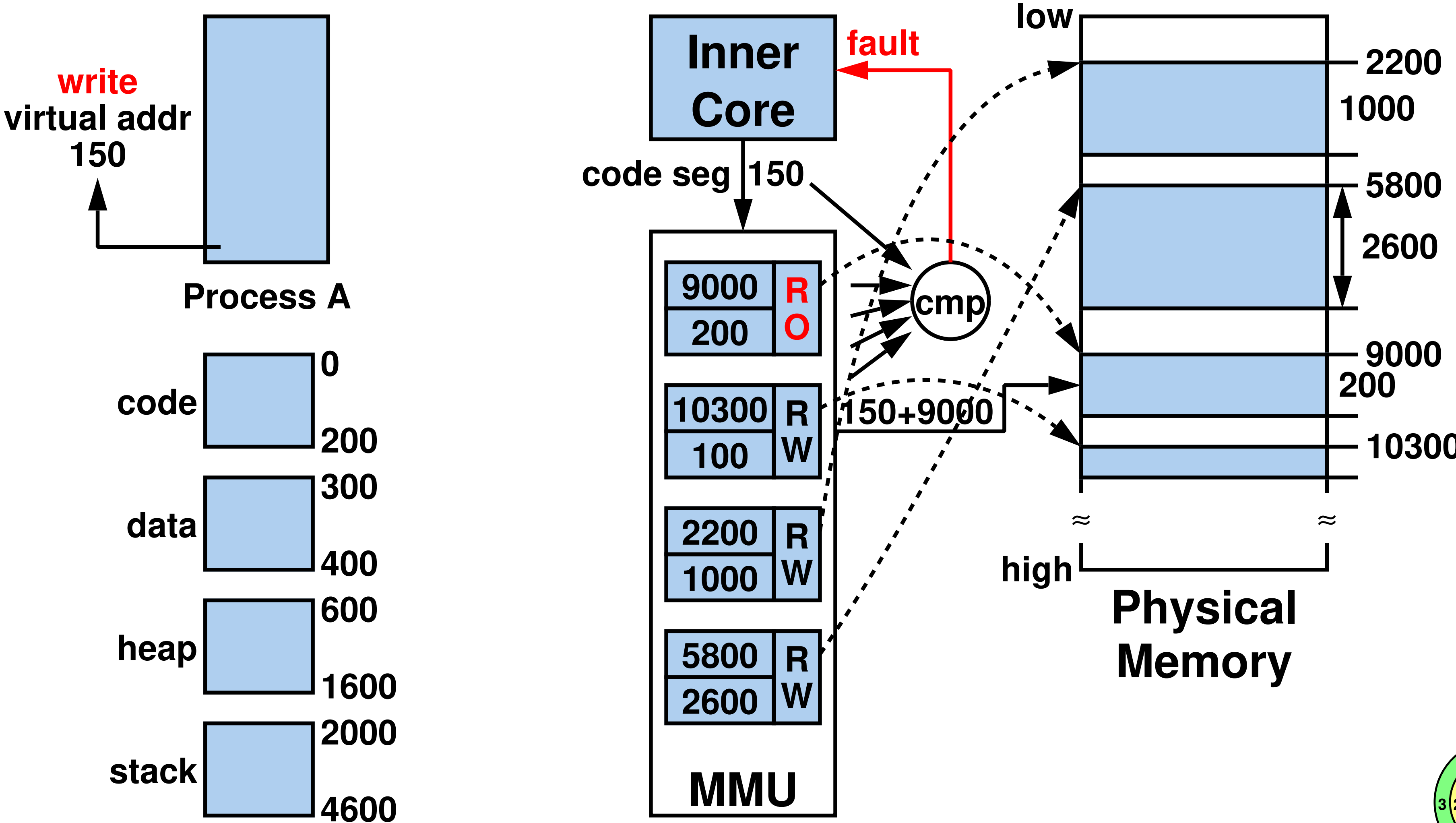
Access Control With Segmentation

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



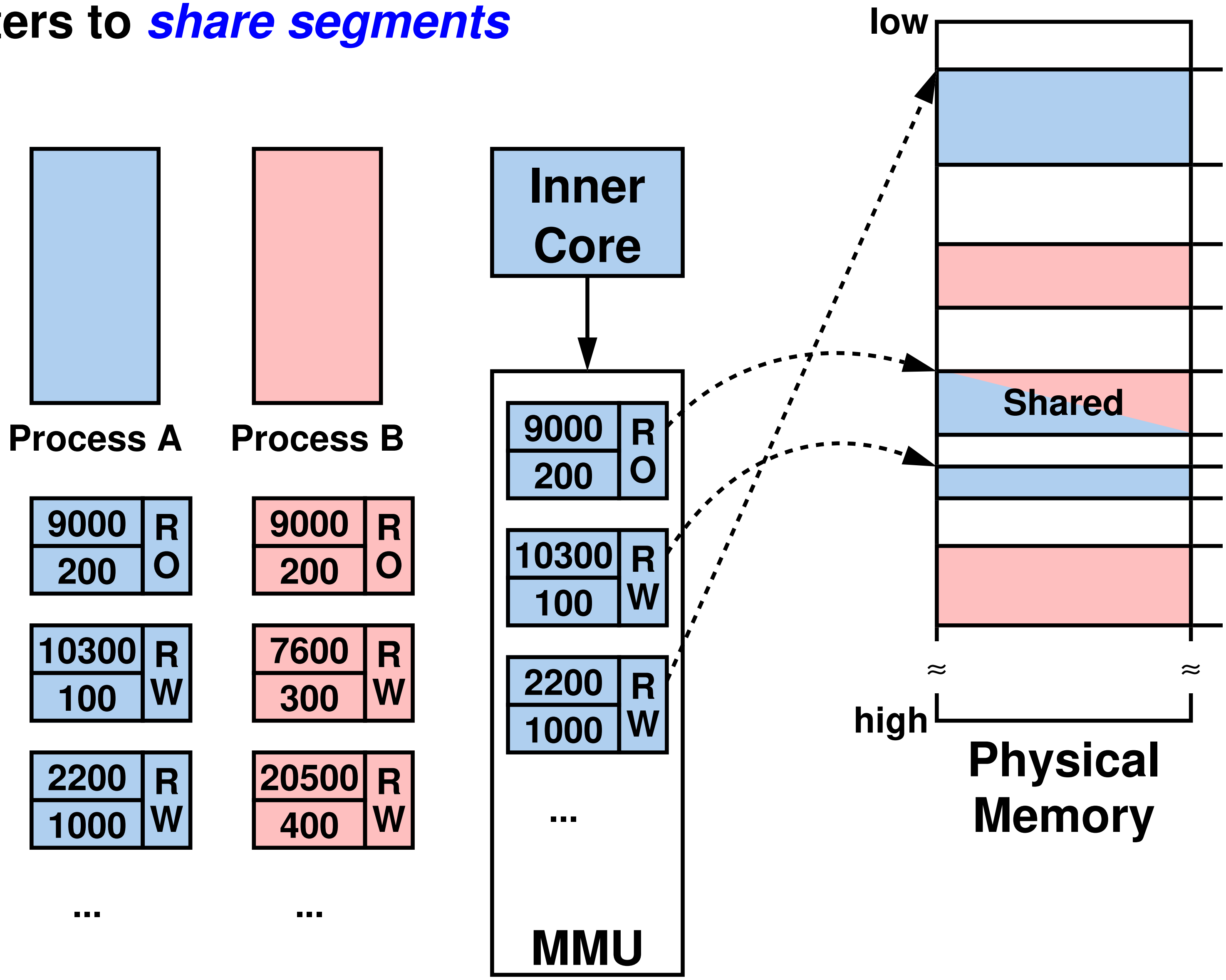
Access Control With Segmentation

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



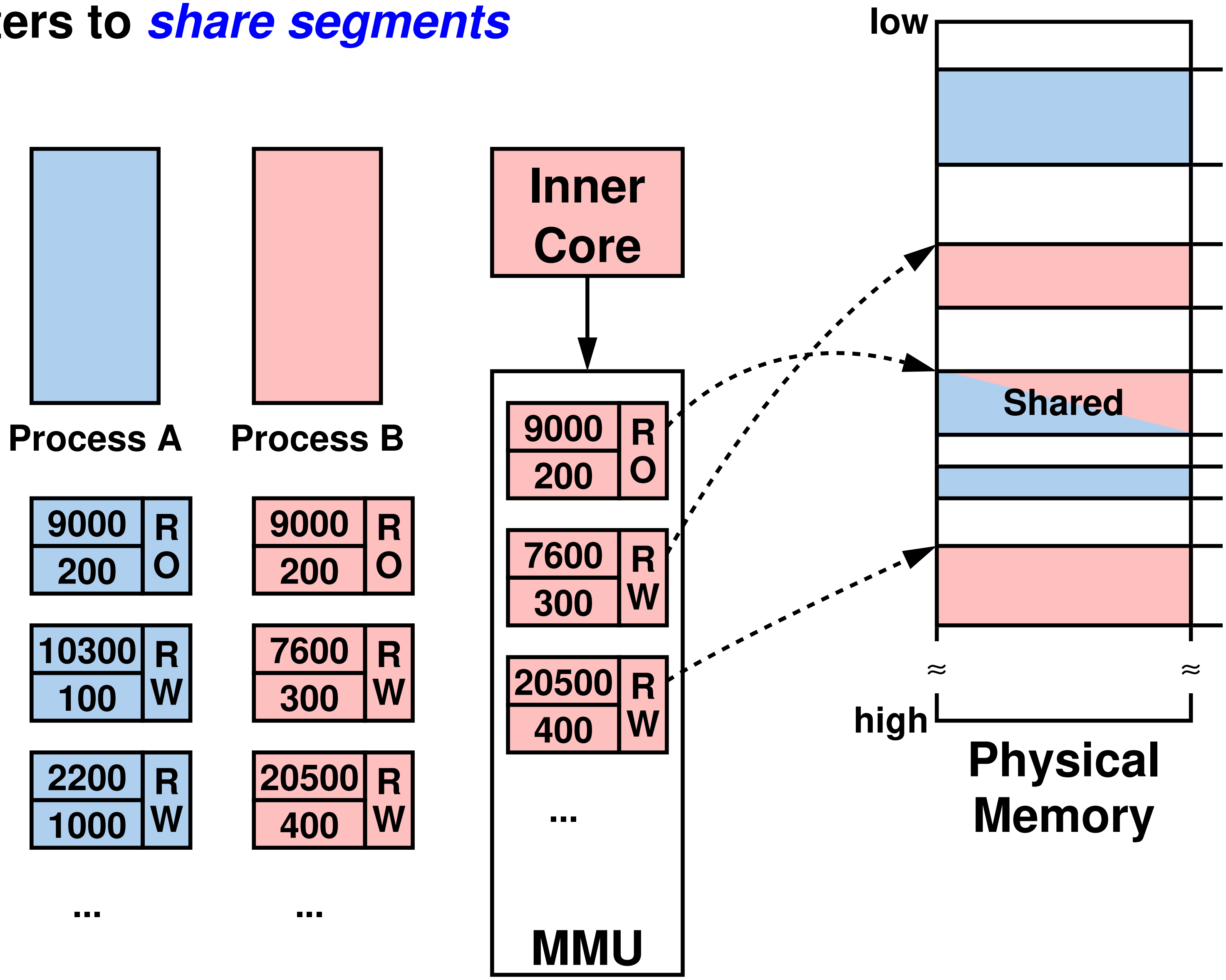
Sharing Segments

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



Sharing Segments

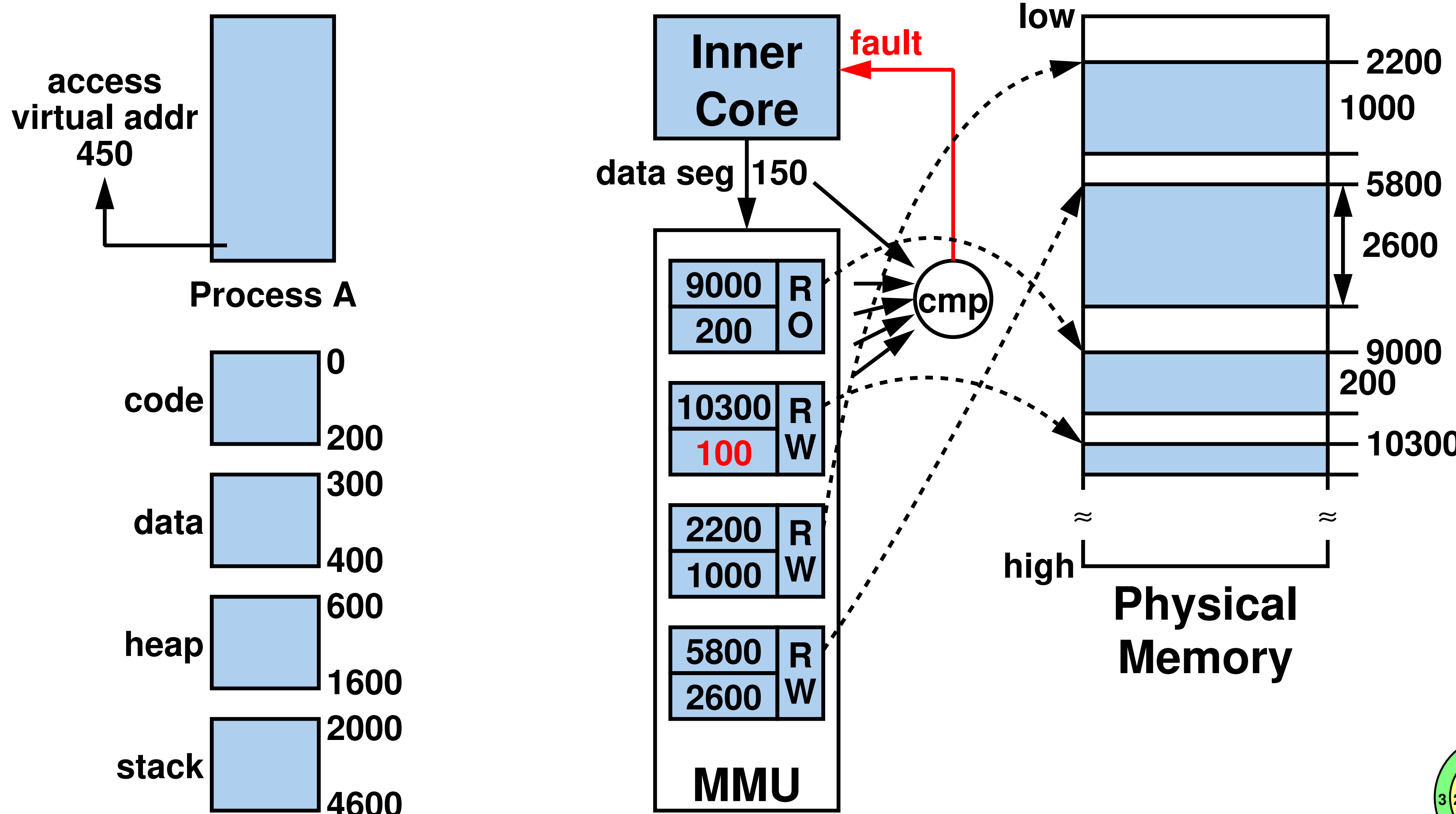
- Can simply setup base and bounds registers to ***share segments***



Segmentation Fault

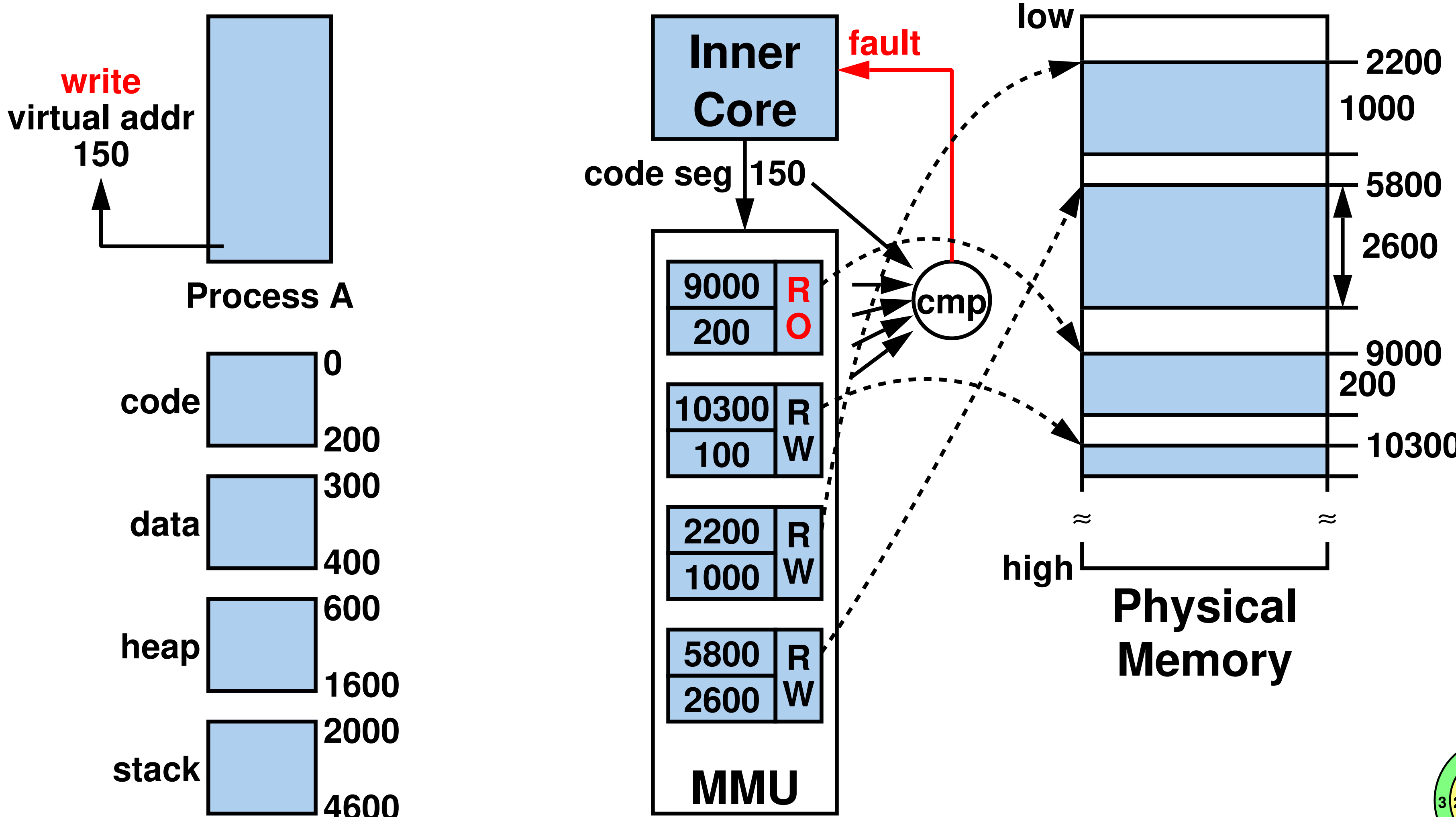
Segmentation fault

- virtual address not within range of any base-bounds registers



Segmentation Fault

- *Segmentation fault*
  - virtual address not within range of any base-bounds registers
  - or access is incompatible

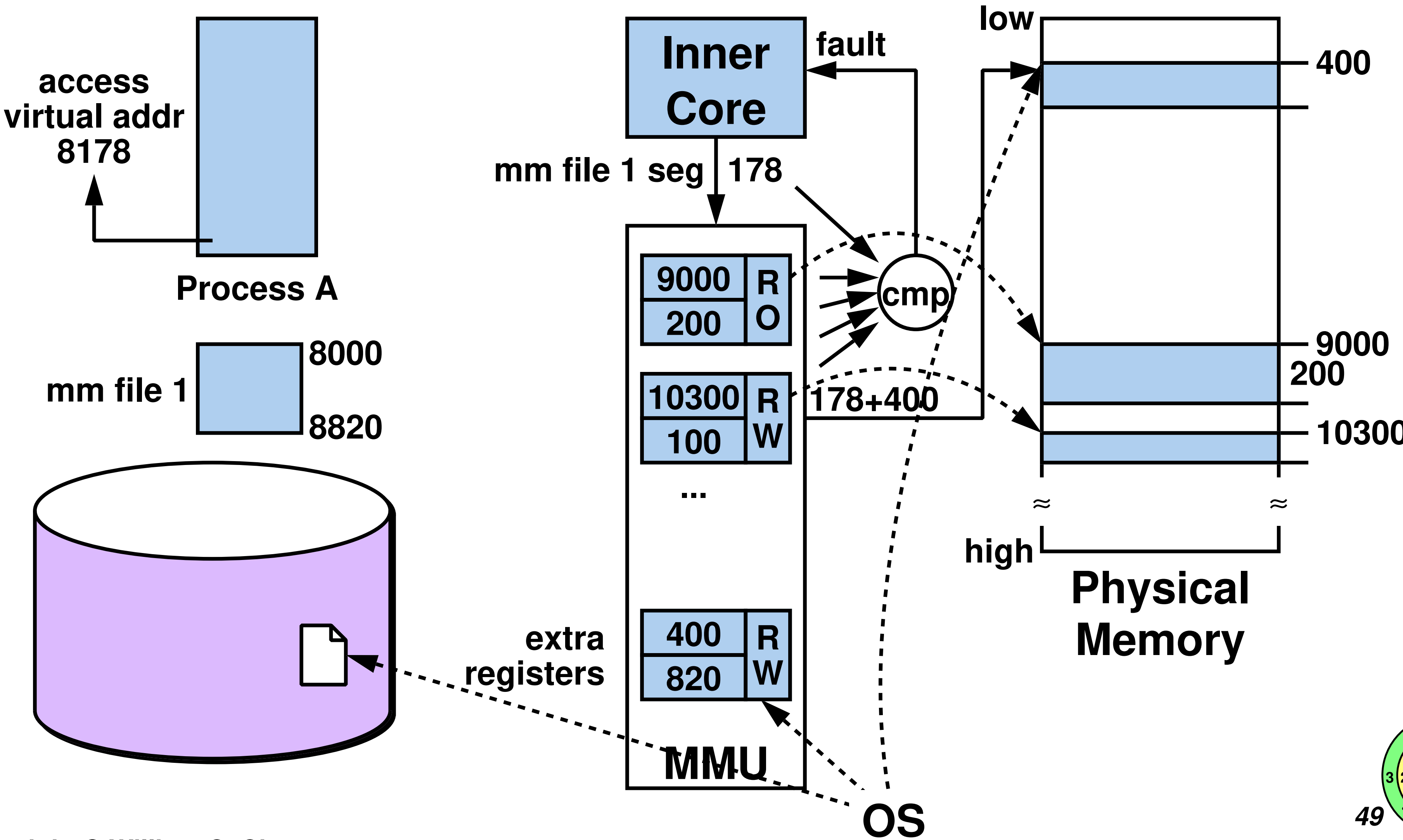


Memory Mapped File

- need more pairs of MMU registers in hardware

Memory Mapped File

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

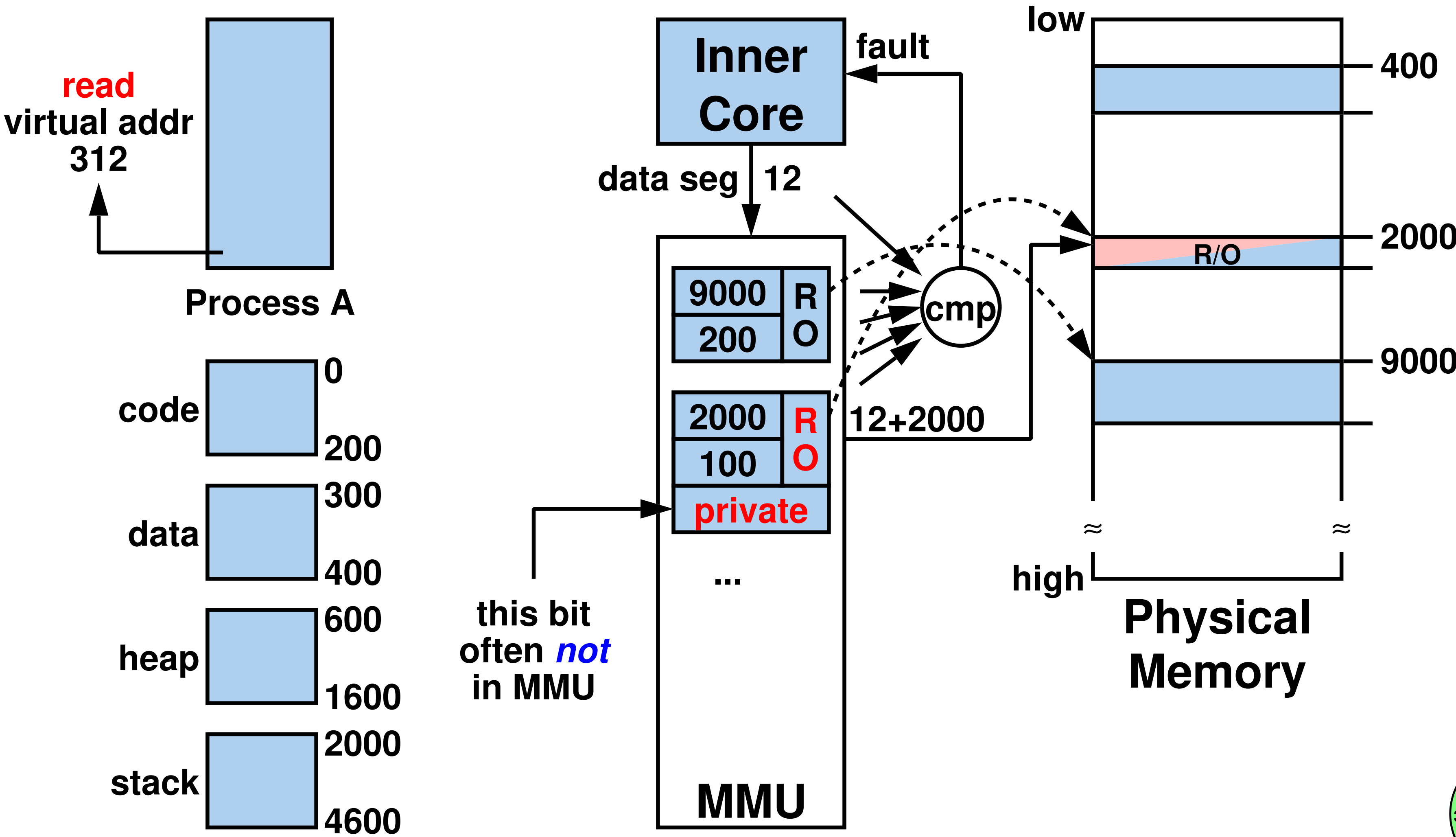


Copy-On-Write

- set R/O bit for *private*, *R/W* memory segment

Copy-on-write (COW):

- a process gets a *private* copy of the segment after a thread in the process performs a *write* for the *first time*

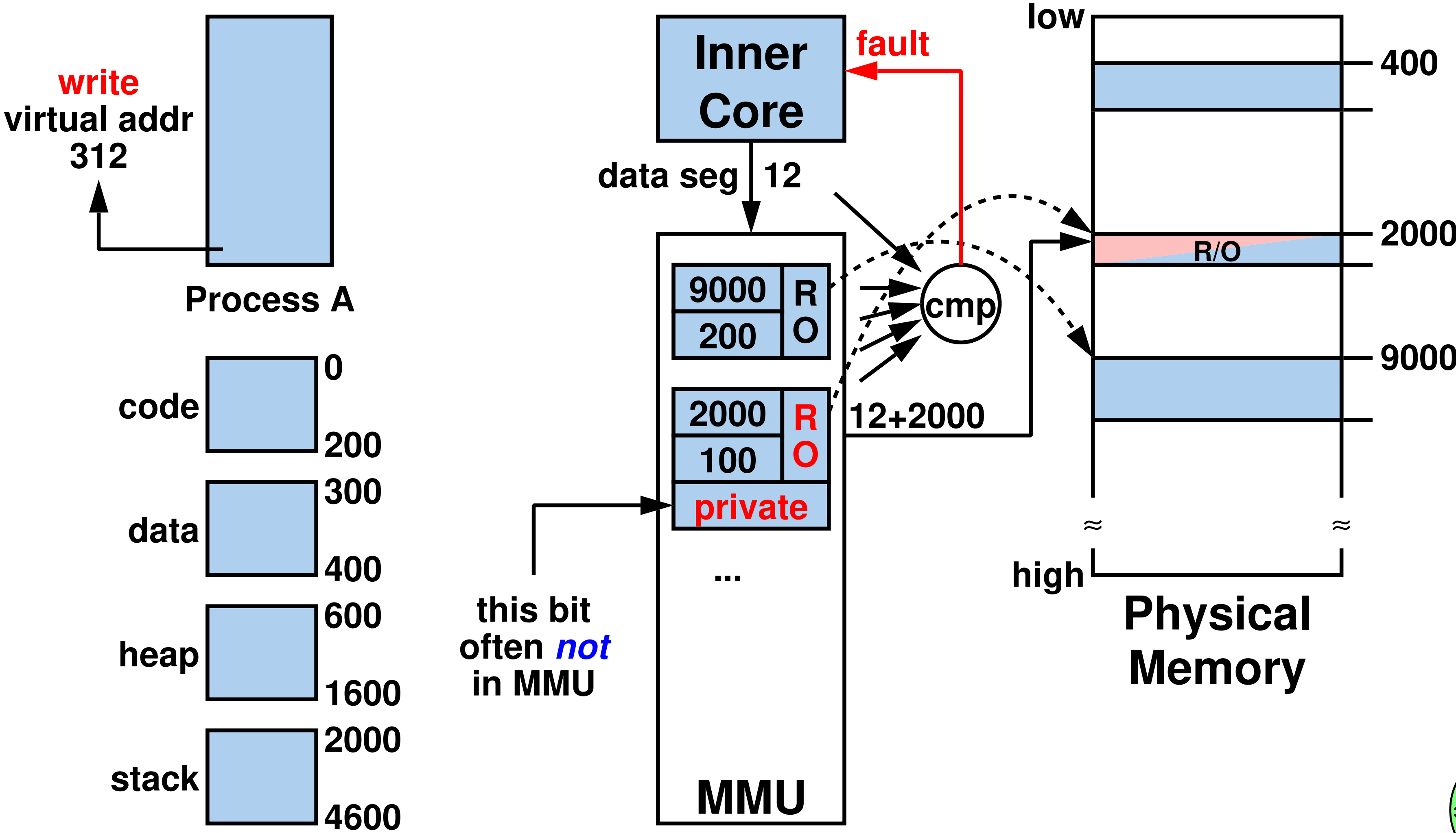


Copy-On-Write

- first time write to this segment traps into OS

Copy-on-write (COW):

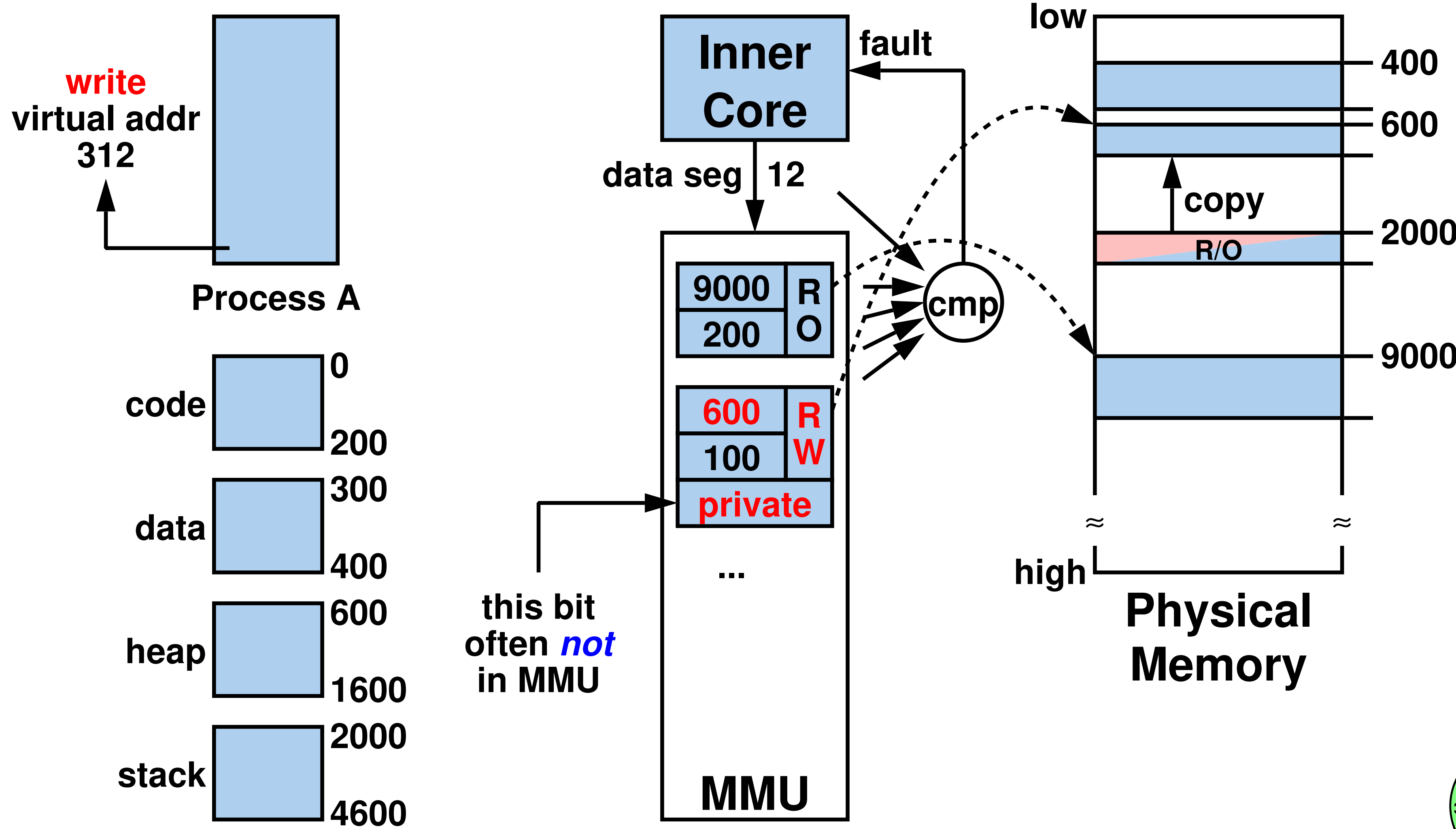
- a process gets a *private* copy of the segment after a thread in the process performs a *write* for the *first time*



Copy-On-Write

Copy-on-write (COW):

- a process gets a *private* copy of the segment after a thread in the process performs a *write* for the *first time*



Copy-On-Write

- future write to this segment will not trap into OS

Copy-on-write (COW):

- a process gets a *private* copy of the segment after a thread in the process performs a *write* for the *first time*