CS 347M (Operating Systems Minor)

Spring 2022

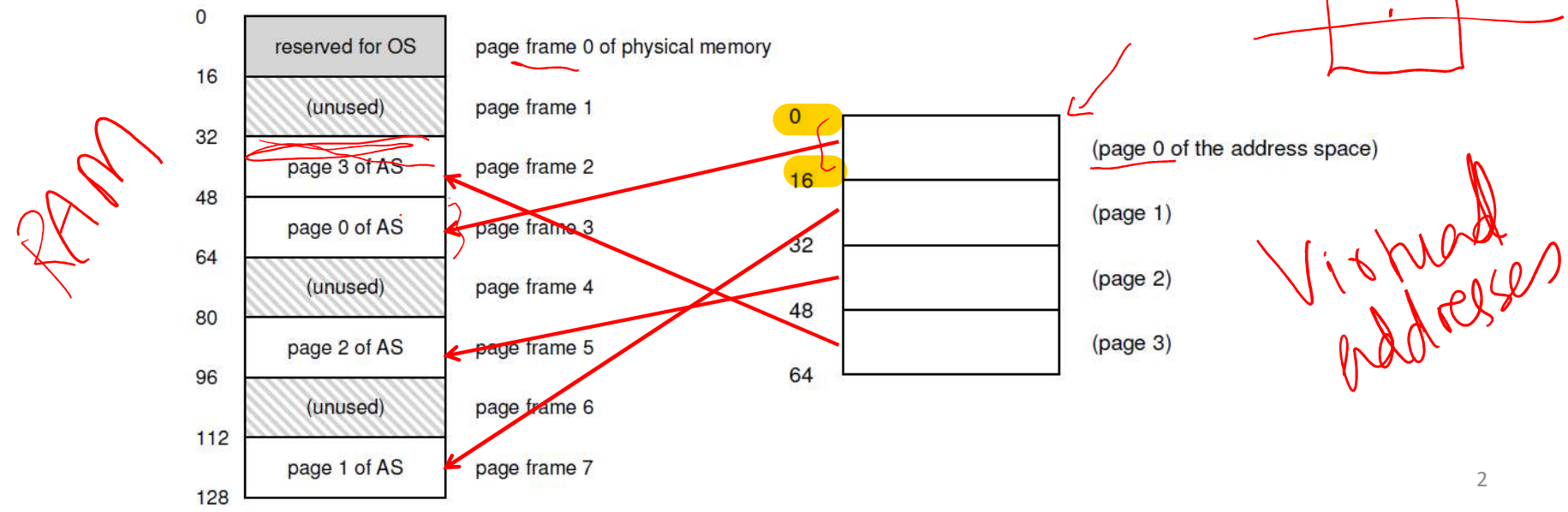
Lecture 8: Paging

Mythili Vutukuru

CSE, IIT Bombay

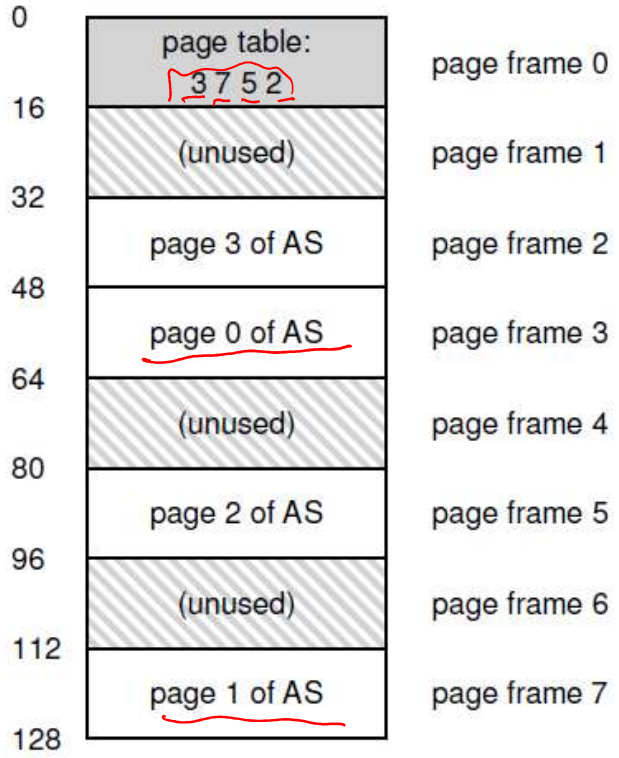
Recap: Paging

- OS allocates memory in fixed size chunks ("pages", typically 4KB)
- Avoids external fragmentation (no small "holes")
- Has internal fragmentation (partially filled pages)
- Virtual address space of process has multiple pages



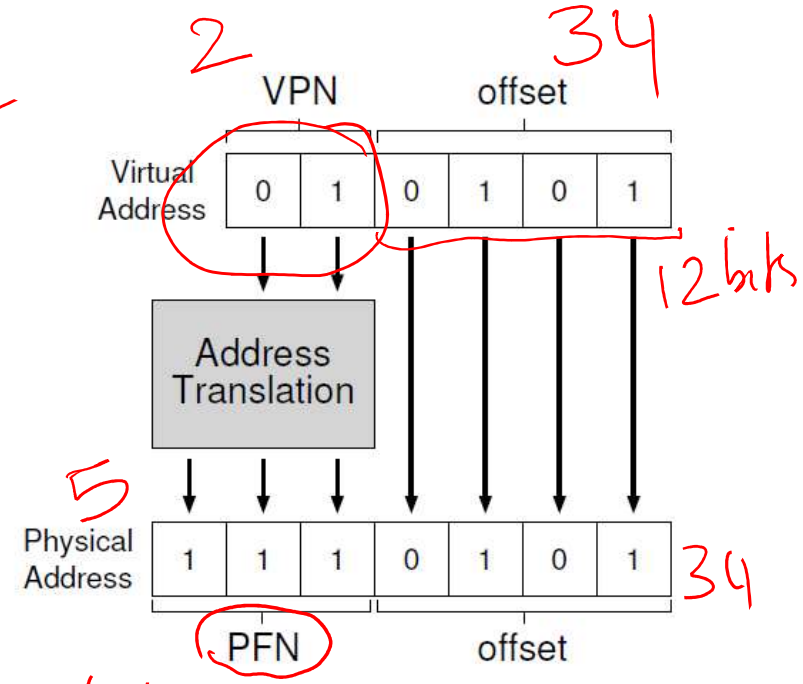
Recap: Page table

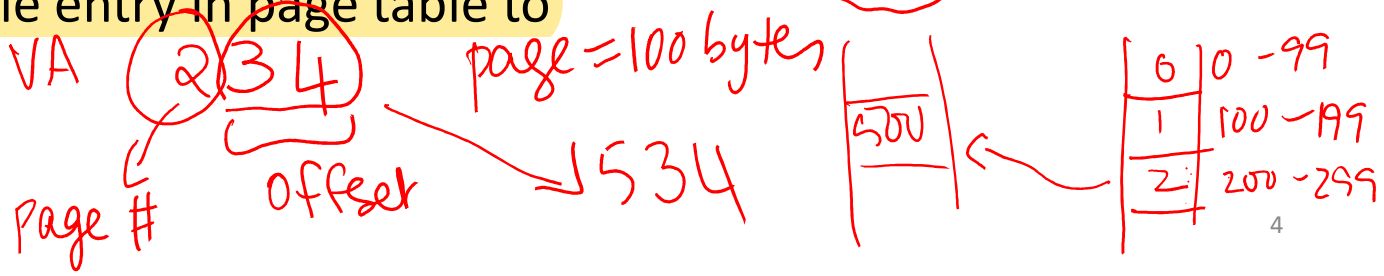
- Per process data structure to help virtual address (VA) to physical address (PA) translation
- Array stores mappings from virtual page number (VPN) to physical frame number (PFN)
  - E.g., VP 0 → PF 3, VP 1 → PF 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address translation
- OS updates page table upon context switch



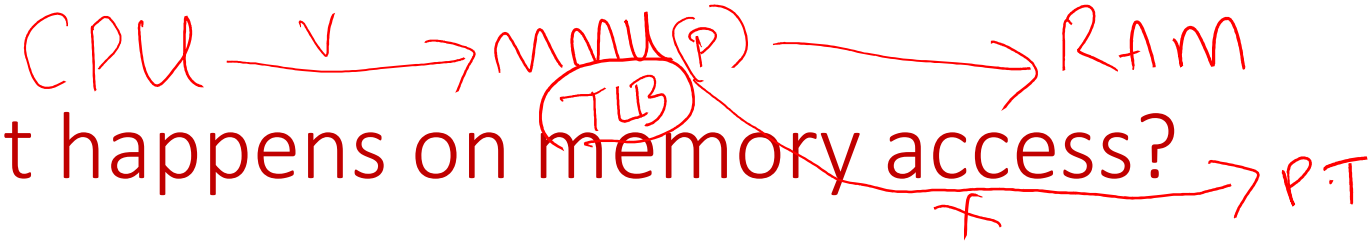
Recap: Address translation in MMU hardware

- Most significant bits of virtual address (VA) give the virtual page number (VPN)
- Least significant bits in VA is offset within page
- Page table array maps VPN to PFN (physical frame number)
- Physical address (PA) is obtained from PFN and offset within a page
- MMU stores (physical) address of start of page table array, not all entries
- MMU accesses suitable entry in page table to translate VA to PA



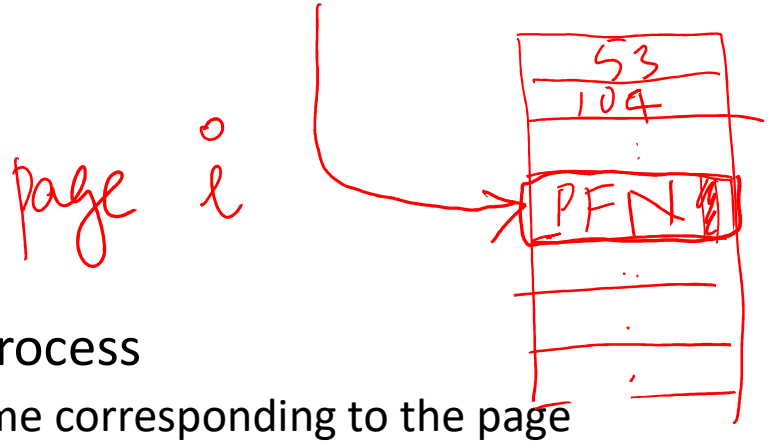


Recap: What happens on memory access?

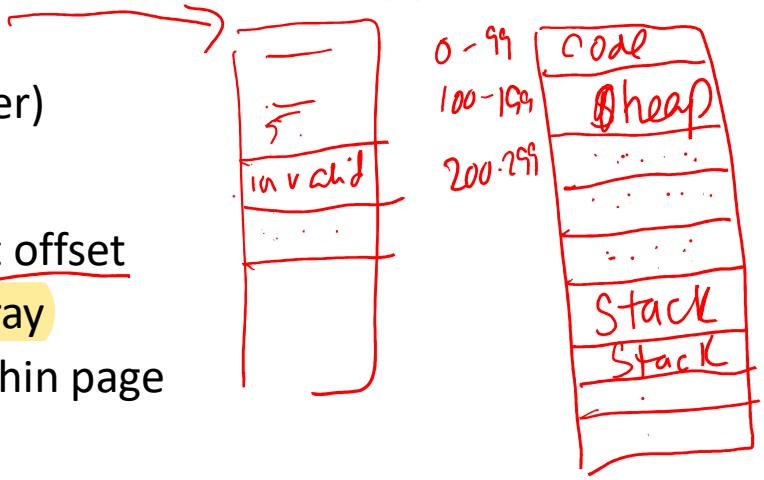


- CPU requests code or data at a virtual address, if not found in CPU caches
- MMU must translate VA to PA before memory access
  - First, access main memory to read page table entry, translate VA to PA
  - Then, access memory to fetch the actual code/data requested by CPU
- To avoid extra memory access for page table, MMU stores cache of recent VA-PA mappings in TLB (Translation Lookaside Buffer)
- If VA found in TLB (TLB hit), memory directly accessed using PA
- If VA not in TLB (TLB miss), MMU accesses memory to read page table, then accesses memory again after computing PA

Structure of page table

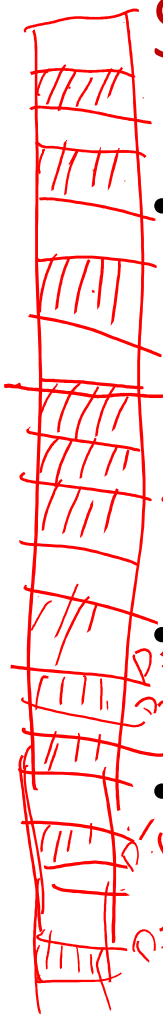


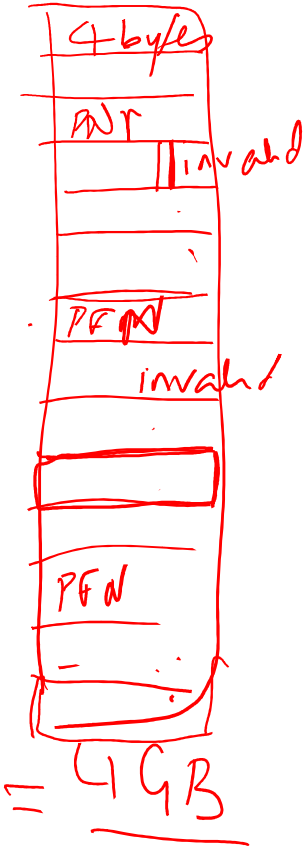
- Array of page table entries, one per page of process
  - Each page table entry "points to" the physical frame corresponding to the page
- i-th page table entry (PTE) contains physical frame number and other details (permissions, status, ..) of i-th page of process
  - Valid: is this page in use by process (not all virtual addresses are used by process)
  - Various permission bits (more later)
  - Other status bits: present, dirty, accessed (more later)
- Address translation using page table
  - 32 bit virtual address = 20 bit page number + 12 bit offset
  - Use 20 bit page number to index into page table array
  - Find frame number for page number, add offset within page



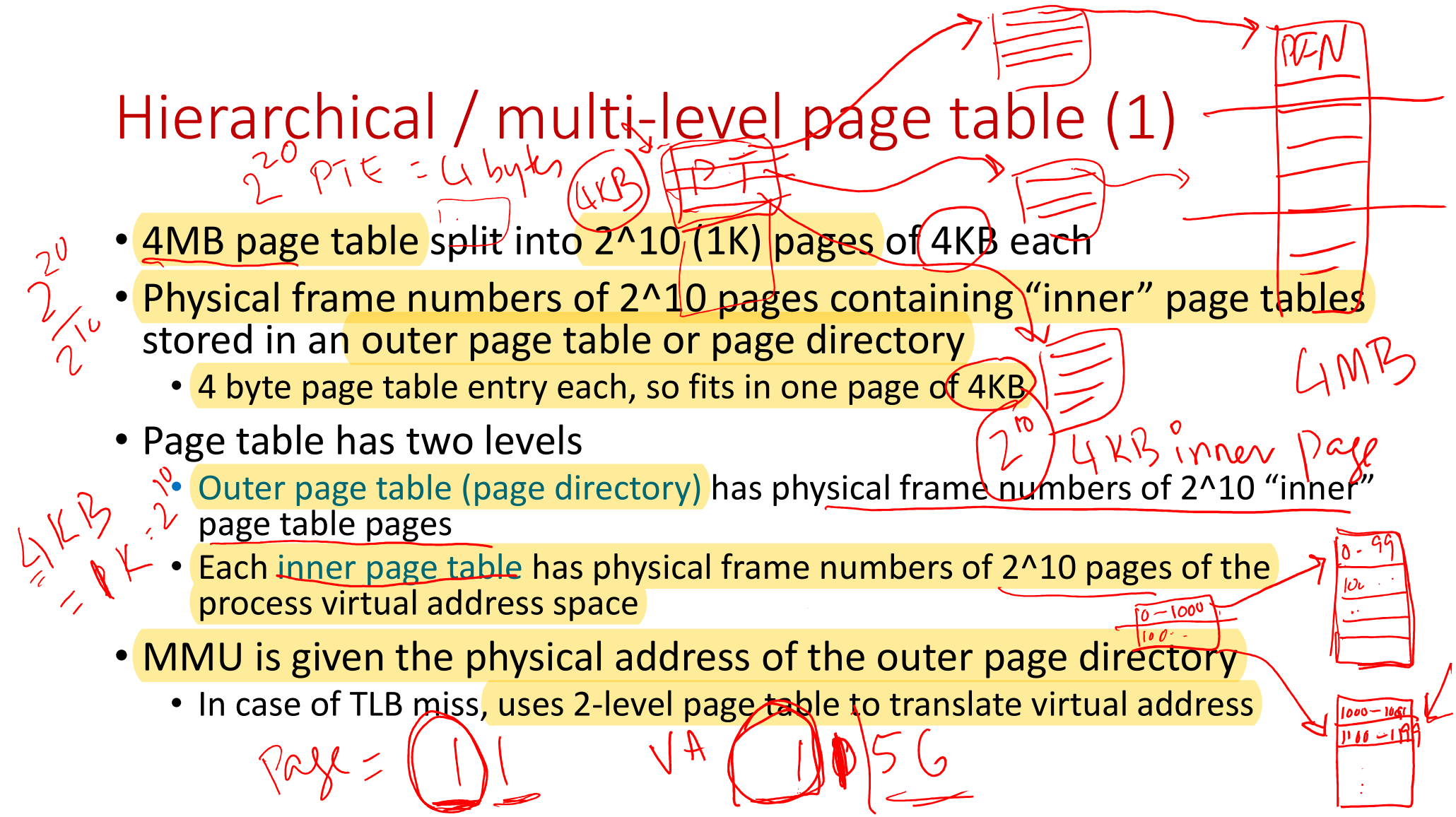
Storing page table in memory

- What is typical size of page table?
  - 32-bit system, 2^32 = 4GB virtual address space
  - Assume page size = 4KB = 2^12
  - Number of pages = (2^32/2^12) = 2^20 = 1M
  - Assume each page table entry is 4 bytes
  - Page table size of one process = 4MB
- How are page tables stored in memory?
  - All memory is only allocated in 4KB chunks
- Solution: split page table into pages (much like memory image), use another page table to keep track of original page table!





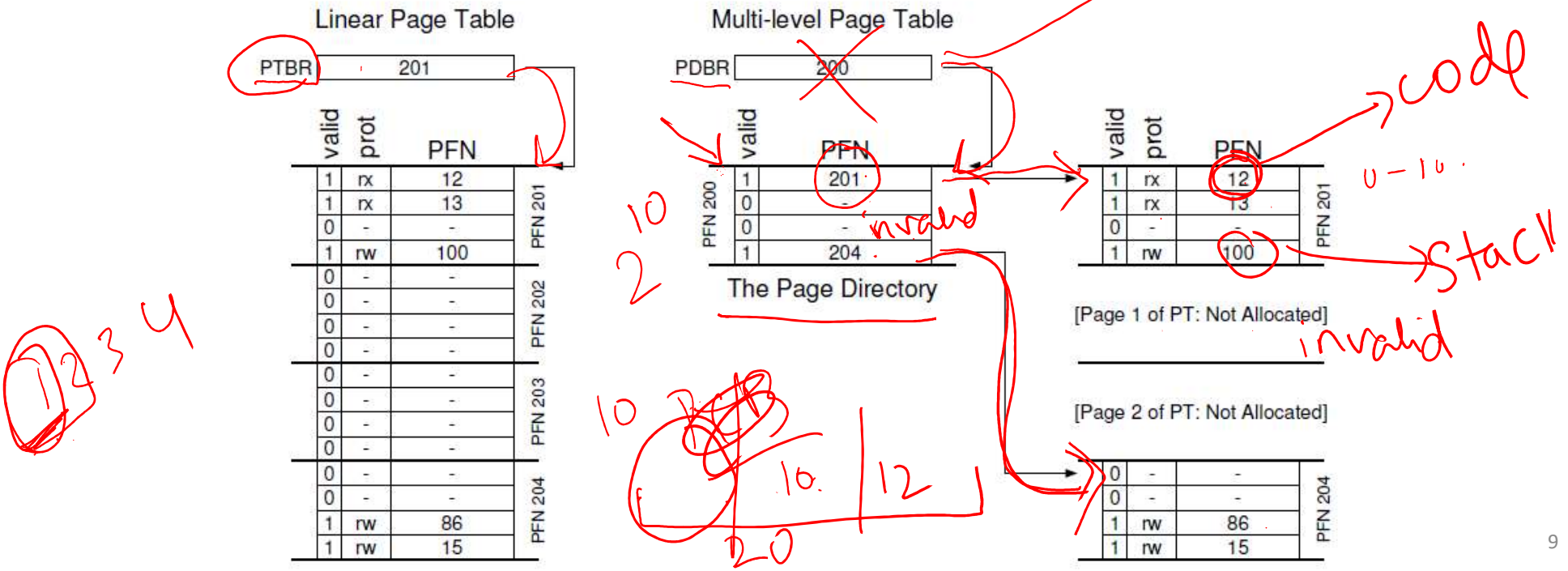
Hierarchical / multi-level page table (1)



- 4MB page table split into 2^10 (1K) pages of 4KB each
- Physical frame numbers of 2^10 pages containing "inner" page tables stored in an outer page table or page directory
  - 4 byte page table entry each, so fits in one page of 4KB
- Page table has two levels
  - Outer page table (page directory) has physical frame numbers of 2^10 "inner" page table pages
  - Each inner page table has physical frame numbers of 2^10 pages of the process virtual address space
- MMU is given the physical address of the outer page directory
  - In case of TLB miss, uses 2-level page table to translate virtual address

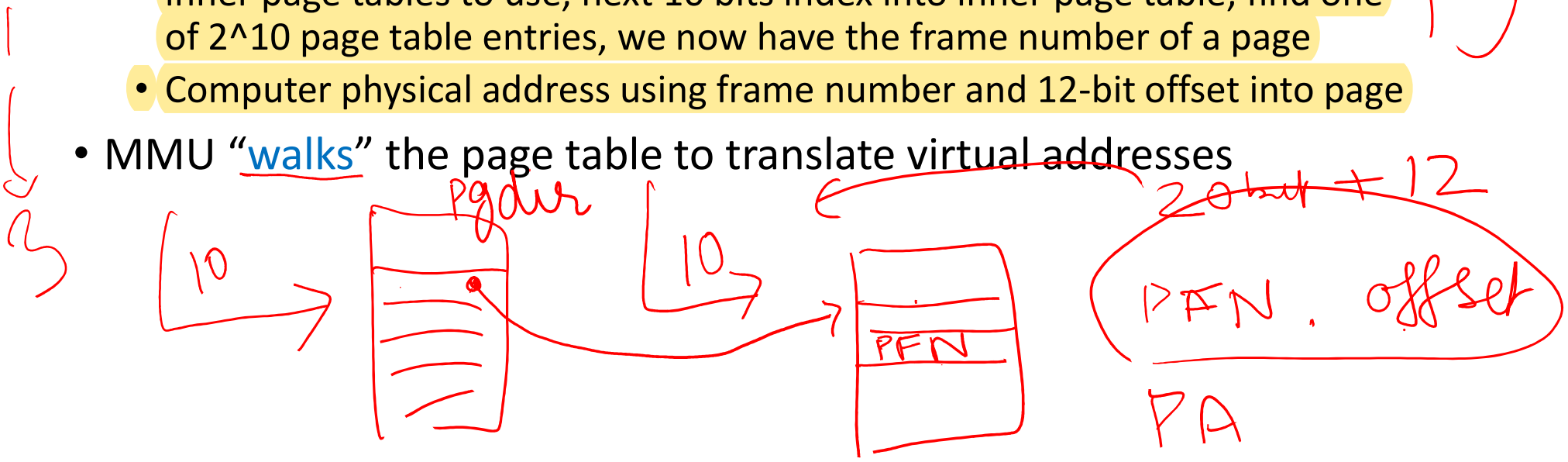
Hierarchical / multi-level page table (2)

- Single level: MMU stores starting address of page table in page table base register
- Multi-level: MMU stores starting address of outer page directory in page directory base register (CR3 register in x86)

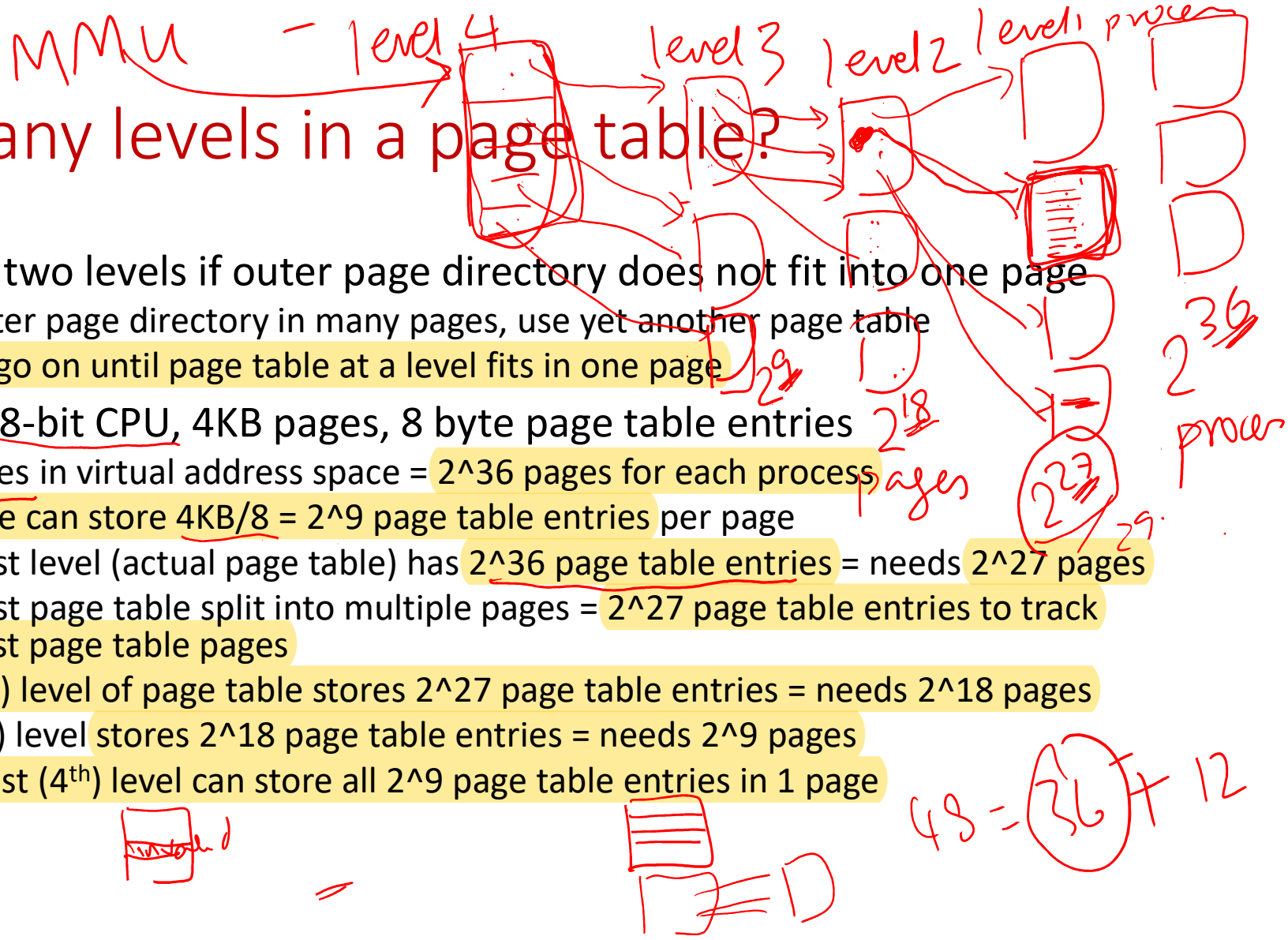


Address translation in 2-level page table

- Virtual address of 32 bits = 20 bit page number + 12 bit offset
  - 20 bits = 10 bit index into page directory, 10 bit index into inner page table
  - Top most 10 bits to index into page directory, identify which one of 2^10 inner page tables to use, next 10 bits index into inner page table, find one of 2^10 page table entries, we now have the frame number of a page
  - Computer physical address using frame number and 12-bit offset into page
- MMU "walks" the page table to translate virtual addresses



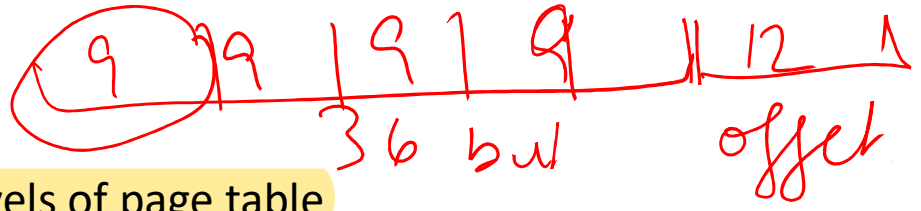
How many levels in a page table?



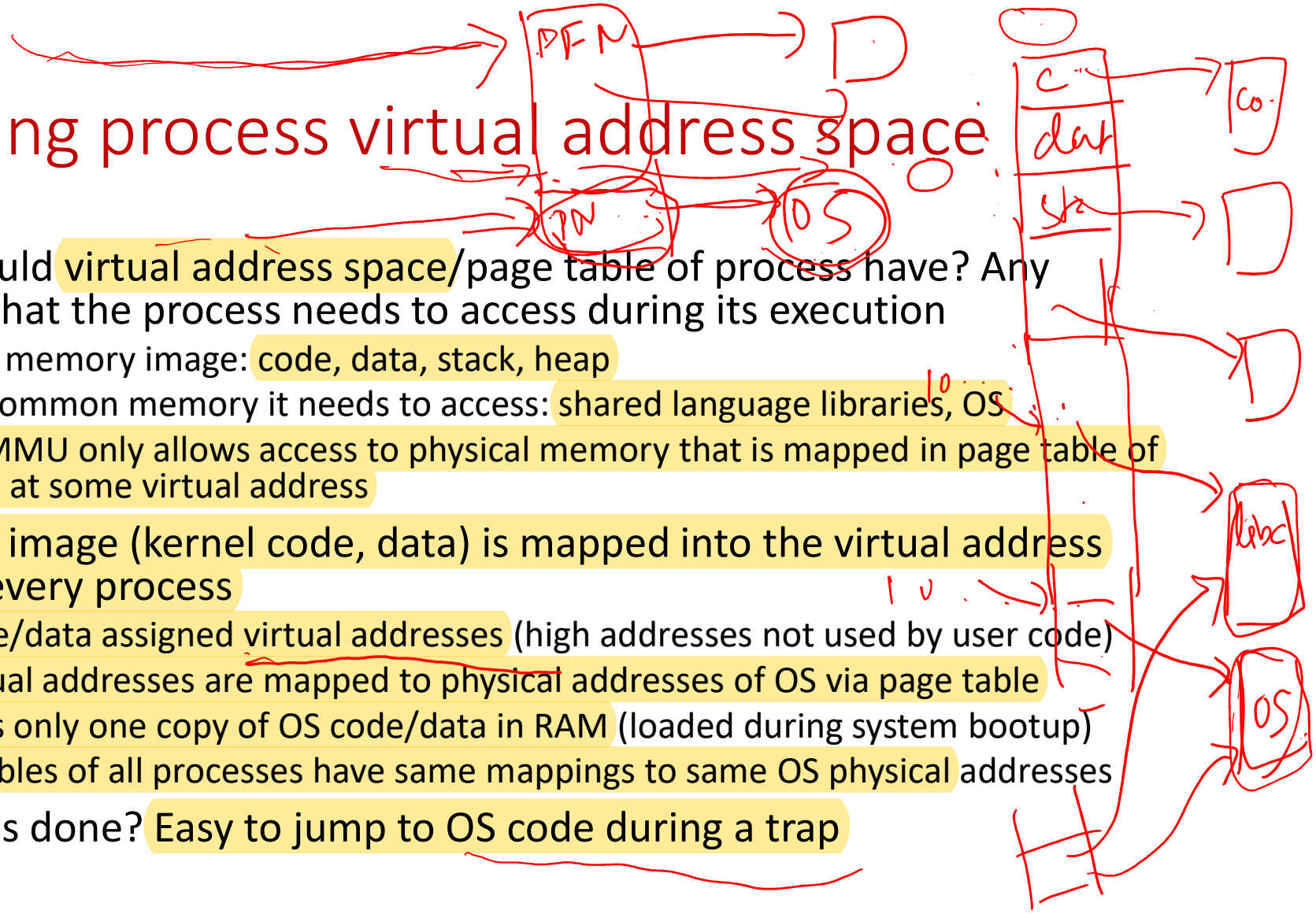
- More than two levels if outer page directory does not fit into one page
  - Store outer page directory in many pages, use yet another page table
  - This can go on until page table at a level fits in one page
- Example: 48-bit CPU, 4KB pages, 8 byte page table entries
  - 2^48 bytes in virtual address space = 2^36 pages for each process
  - Each page can store 4KB/8 = 2^9 page table entries per page
  - Innermost level (actual page table) has 2^36 page table entries = needs 2^27 pages
  - Innermost page table split into multiple pages = 2^27 page table entries to track innermost page table pages
  - Next (2nd) level of page table stores 2^27 page table entries = needs 2^18 pages
  - Next (3rd) level stores 2^18 page table entries = needs 2^9 pages
  - Outermost (4th) level can store all 2^9 page table entries in 1 page

Address translation in multi-level page table

- Example: 48-bit CPU, 4KB pages, 8 byte page table entries
  - 4 level page table required
  - Outermost 4th level page directory has 2^9 entries, containing frame numbers of 3rd level page table pages
  - Each page of 3rd level page table has frame numbers of 2^9 2nd level page table pages
  - And so on….
- How to translate VA to PA?
  - 48-bit VA = 36 bits + 12 bit offset
  - 36 bits = 9 bit offset into each of the 4 levels of page table
- MMU has to access 4 different memory locations for 4 levels of page table, in order to translate one VA to PA
  - MMU page table walks become even longer, TLB hit rate is critical



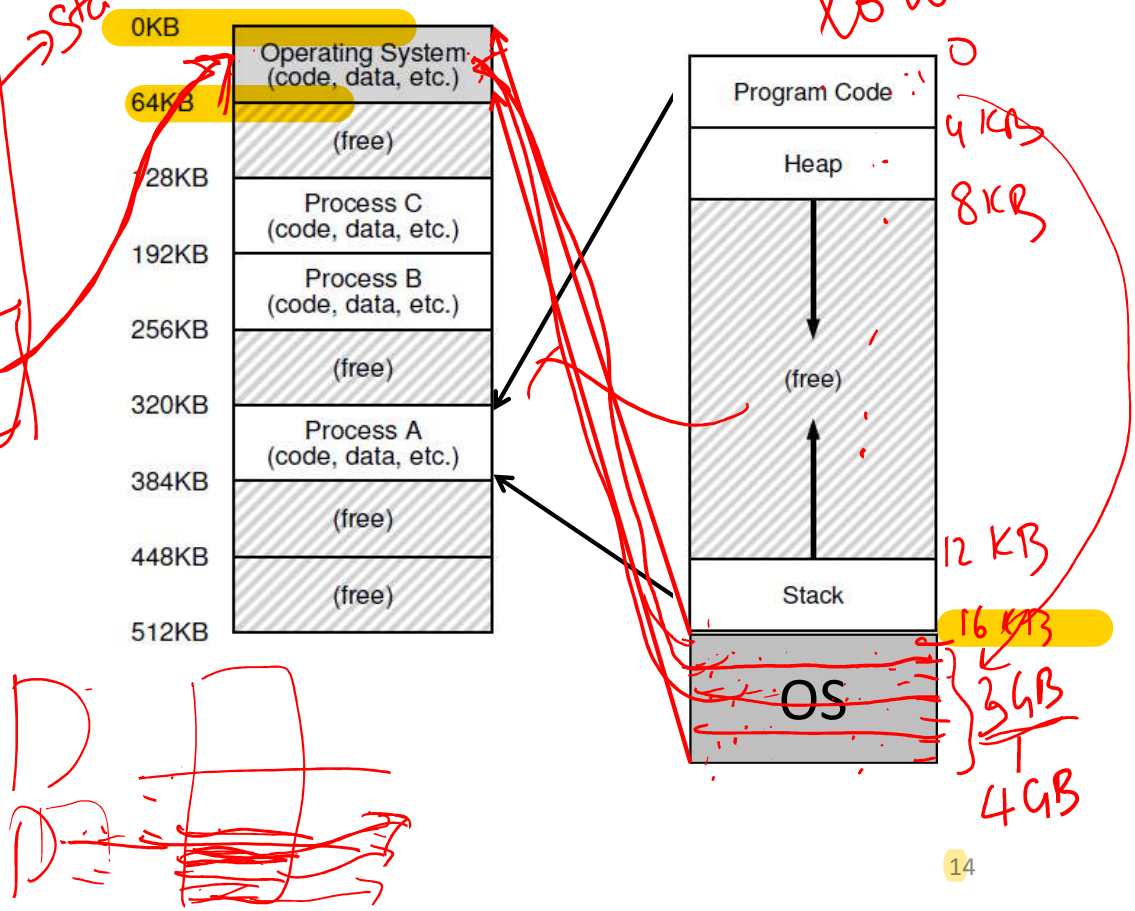
Revisiting process virtual address space



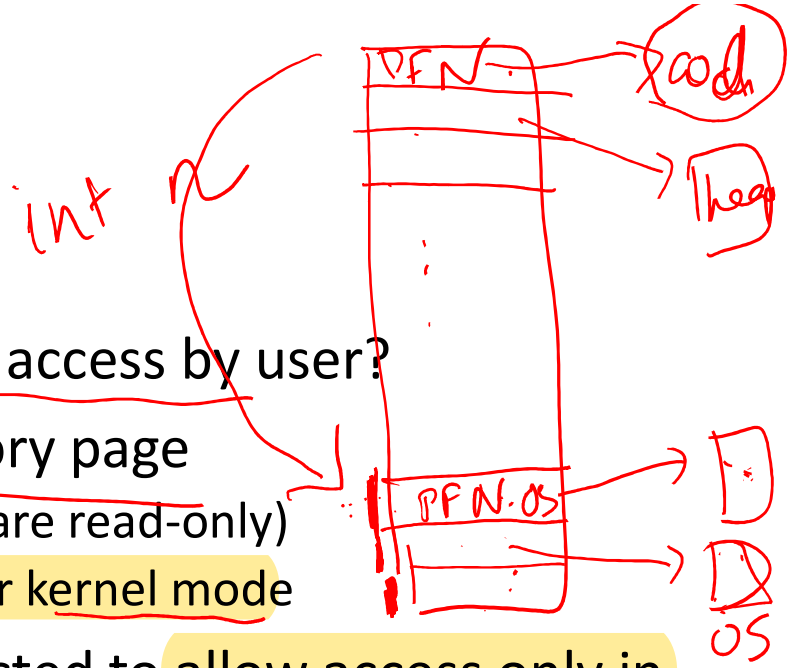
- What should virtual address space/page table of process have? Any memory that the process needs to access during its execution
  - Its own memory image: code, data, stack, heap
  - Other common memory it needs to access: shared language libraries, OS
  - Why? MMU only allows access to physical memory that is mapped in page table of process at some virtual address
- OS binary image (kernel code, data) is mapped into the virtual address space of every process
  - OS code/data assigned virtual addresses (high addresses not used by user code)
  - OS virtual addresses are mapped to physical addresses of OS via page table
  - There is only one copy of OS code/data in RAM (loaded during system bootup)
  - Page tables of all processes have same mappings to same OS physical addresses
- Why is this done? Easy to jump to OS code during a trap

A subtle point: OS is part of address space of every process

- OS is not a separate process with its own address space
- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., like a library)
- Page table of every process maps the OS addresses to OS code in main memory
- Only one copy of OS code in memory



Isolation and security



- How is OS code/data protected from illegal access by user?
- Page table has permissions for every memory page
  - Whether read/write or read-only (code pages are read-only)
  - Whether page can be accessed in user mode or kernel mode
- Page table mappings for OS code are protected to allow access only in kernel mode
  - User program in user mode cannot jump to high virtual addresses of OS code
  - CPU in kernel mode (after trap instruction) can access OS code/data
- MMU ensures that user programs cannot access any memory beyond what is visible in the user-accessible part of virtual address space