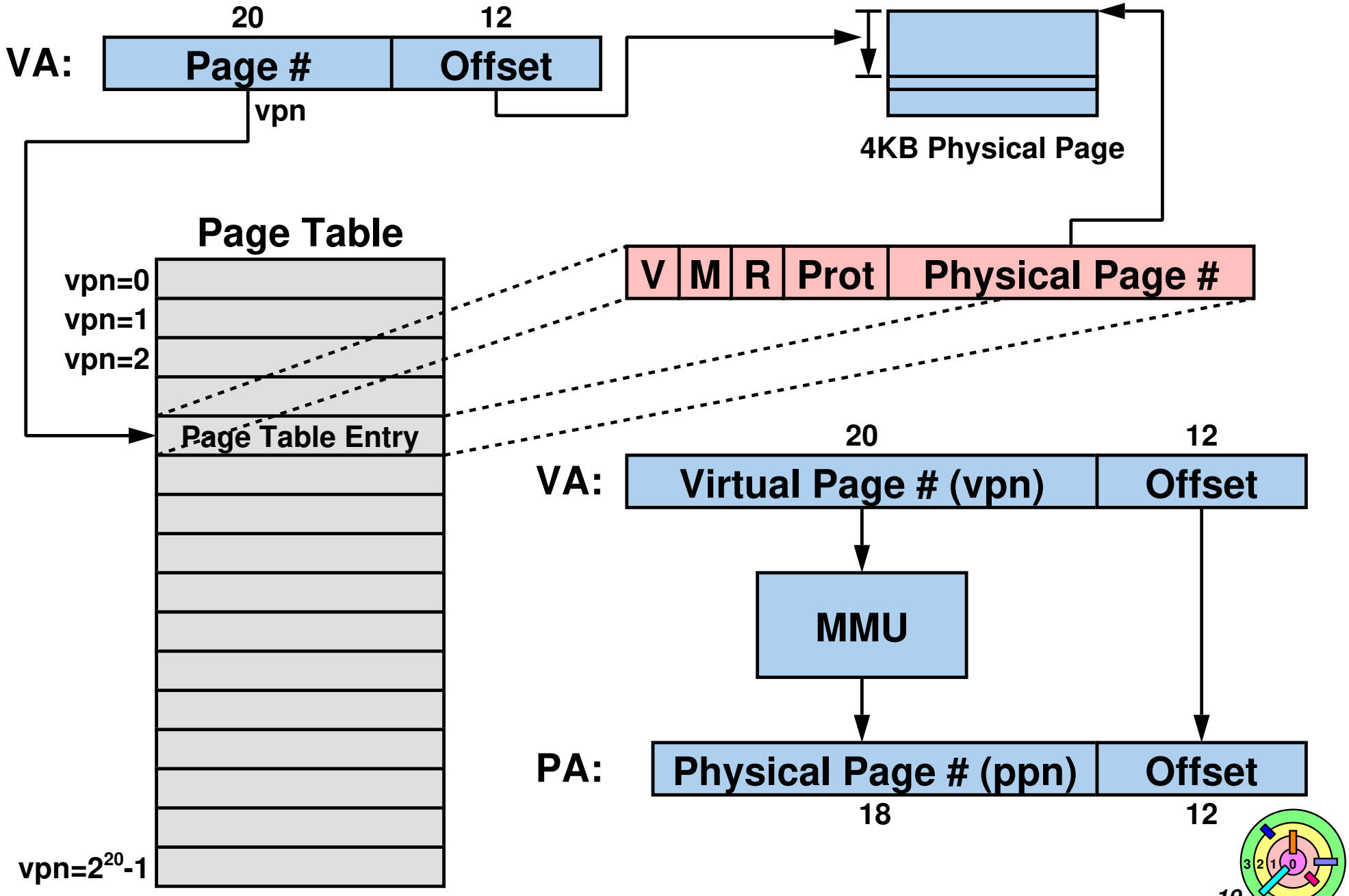
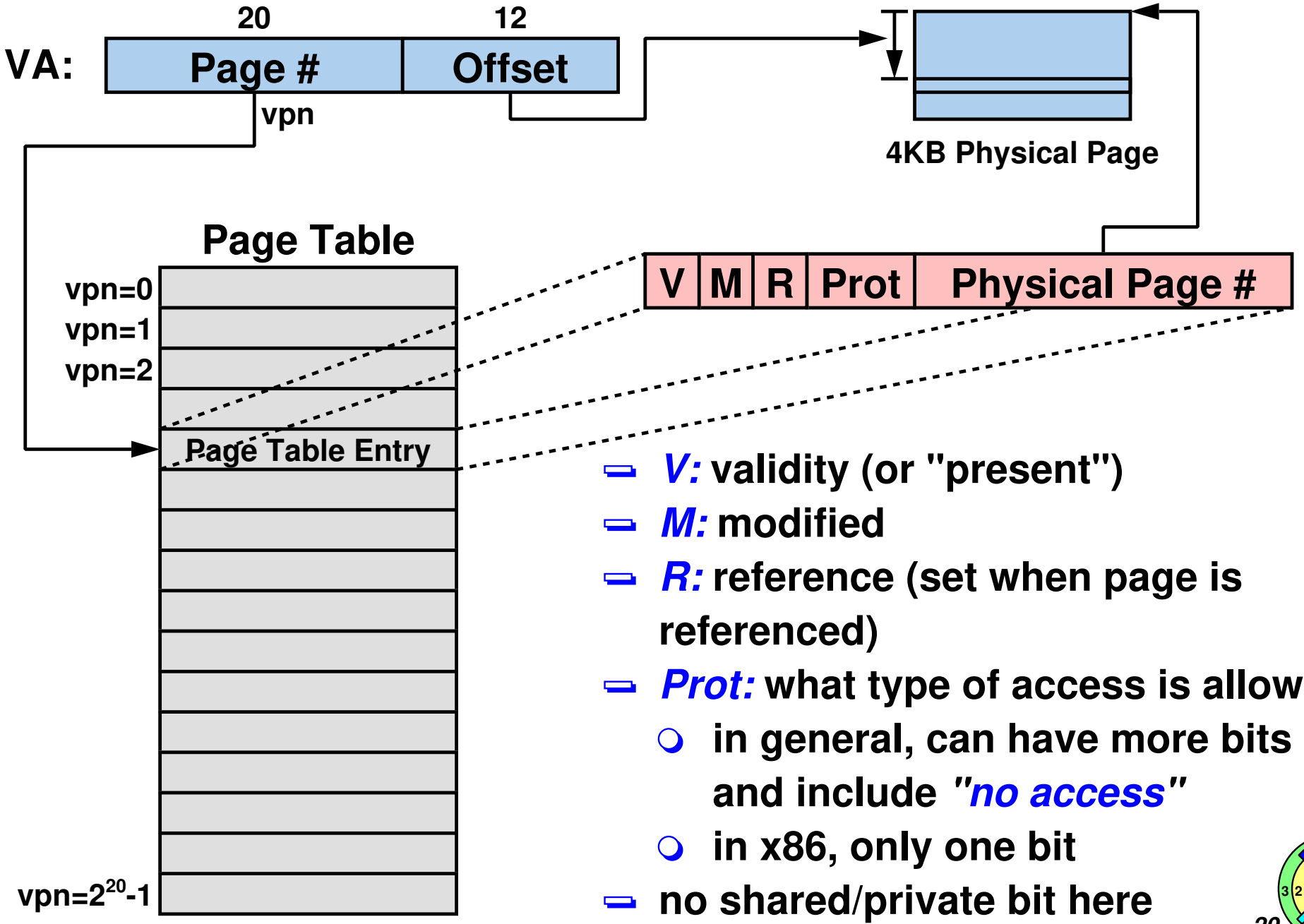
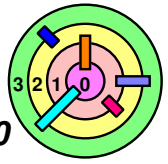
Basic (Two-level) Page Tables



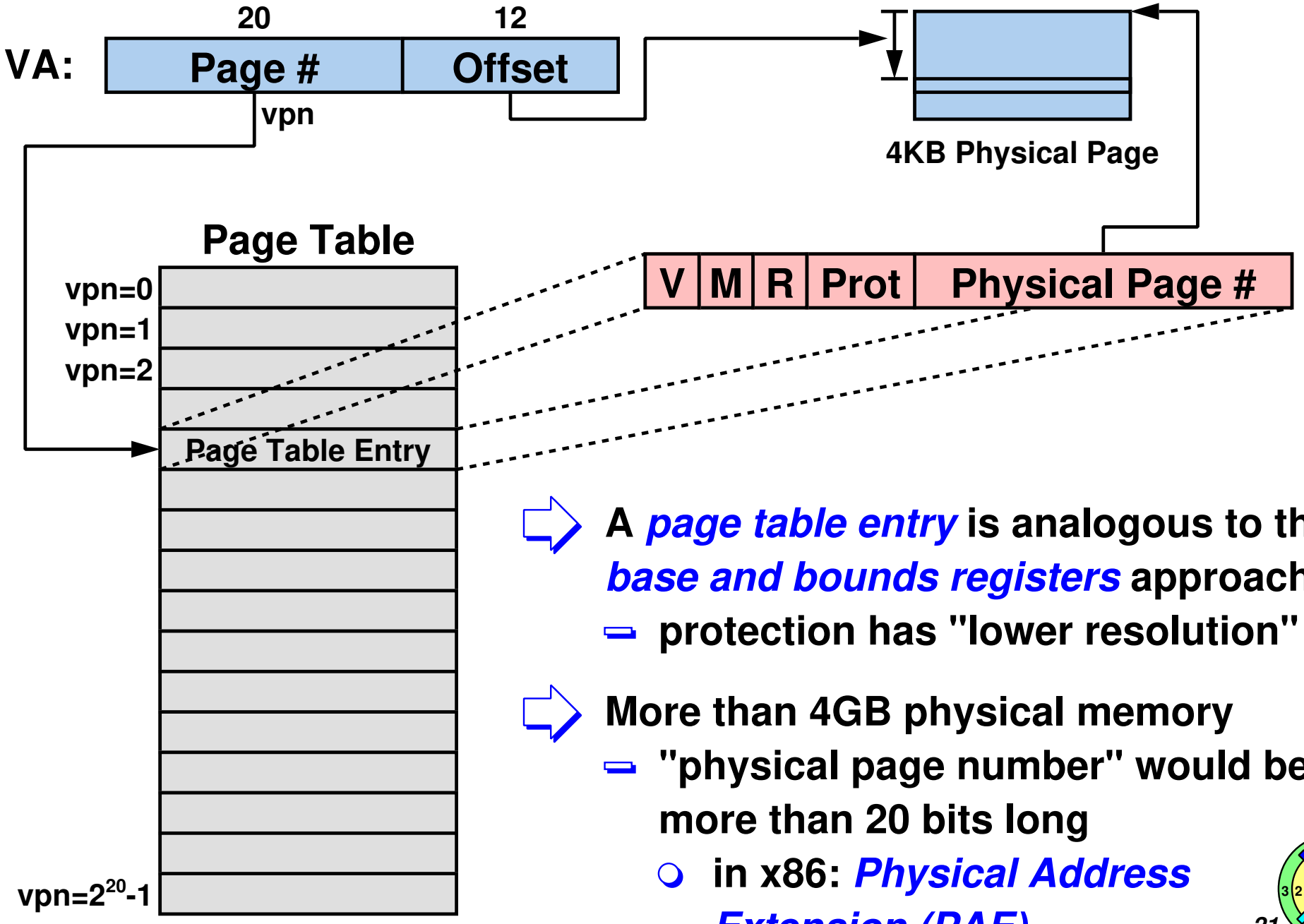
Basic (Two-level) Page Tables



- *V:* validity (or "present")
- *M:* modified
- *R:* reference (set when page is referenced)
- *Prot:* 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

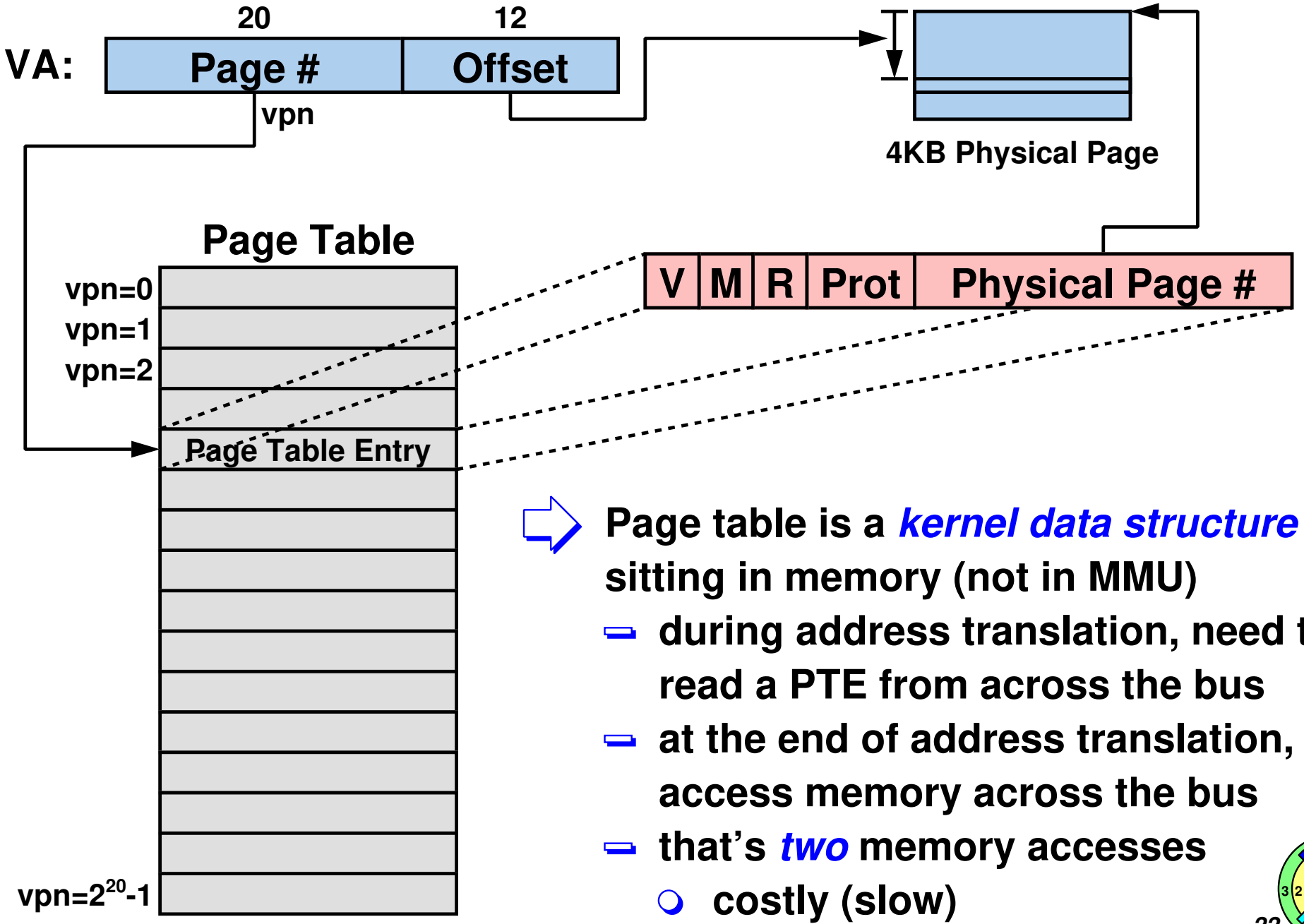


Basic (Two-level) Page Tables



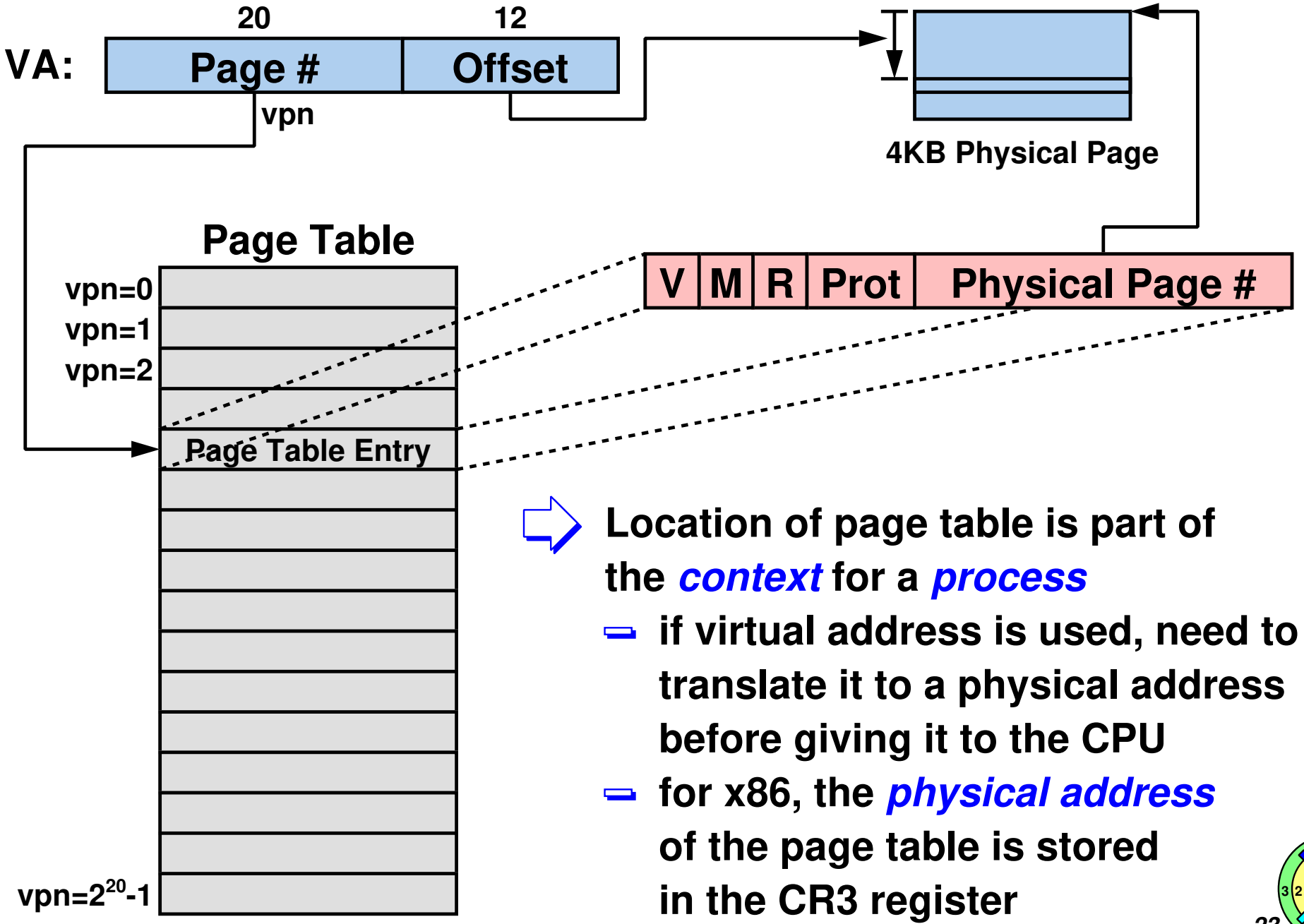
- A *page table entry* is analogous to the *base and bounds registers* approach
  - protection has "lower resolution" here
- More than 4GB physical memory
  - "physical page number" would be more than 20 bits long
    - in x86: *Physical Address Extension (PAE)*

Basic (Two-level) Page Tables

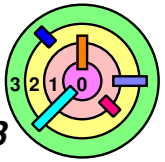


- Page table is a ***kernel data structure*** sitting in memory (not in MMU)
  - during address translation, need to read a PTE from across the bus
  - at the end of address translation, access memory across the bus
  - that's ***two*** memory accesses
    - costly (slow)

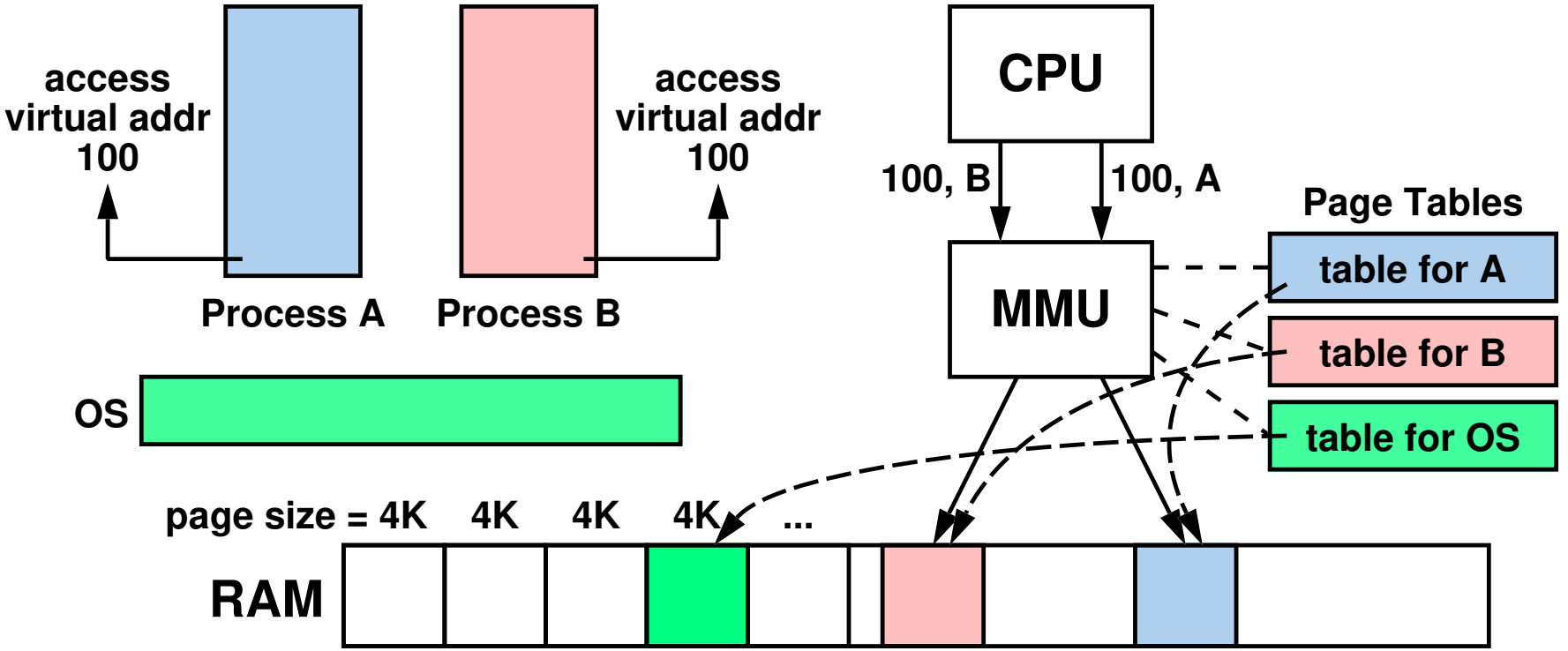
Basic (Two-level) Page Tables



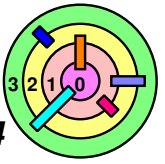
- Location of page table is part of the *context* for a *process*
  - if virtual address is used, need to translate it to a physical address before giving it to the CPU
  - for x86, the *physical address* of the page table is stored in the CR3 register



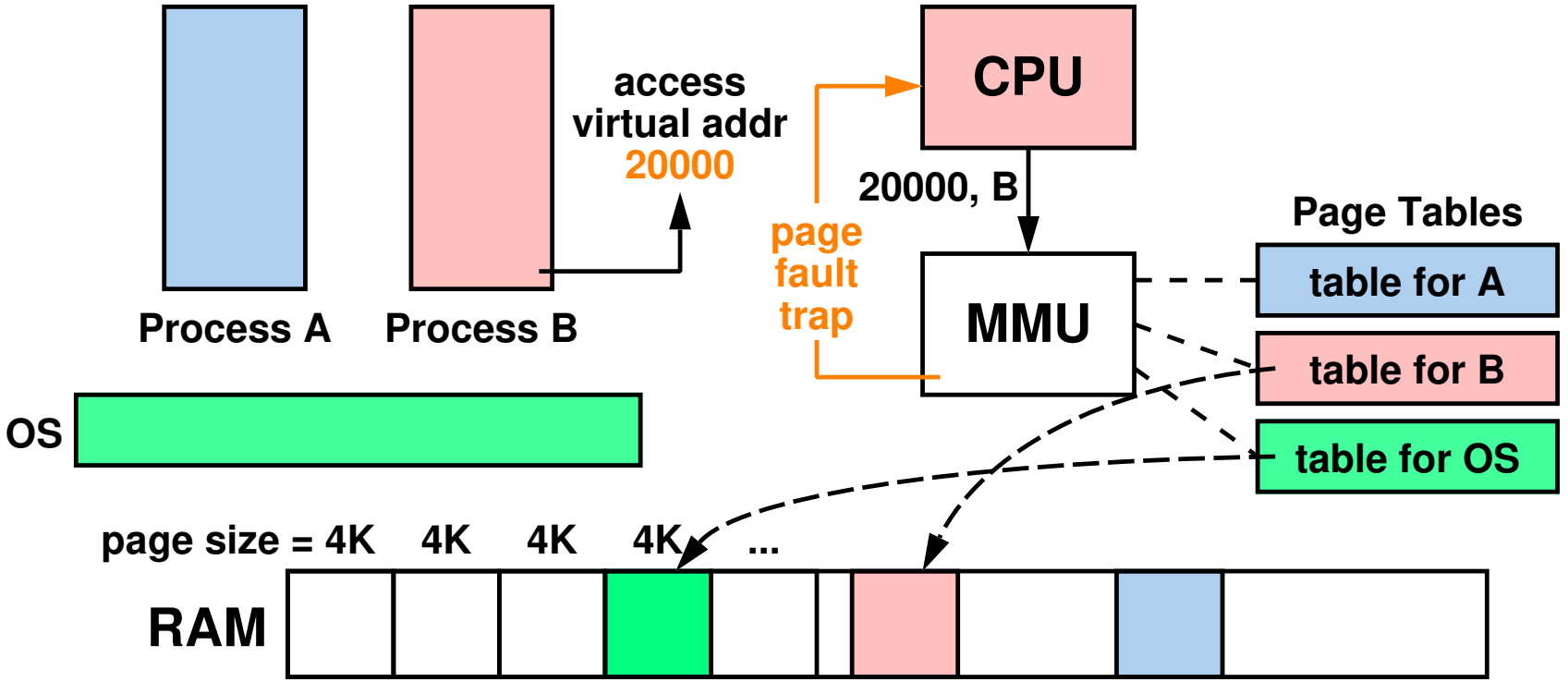
Page Table



- A page table is associated with *each process*
  - OS may have its own page table
- MMU contains the hardware that can perform *address translation* to map virtual address to physical address
  - MMU got turned on some time during boot

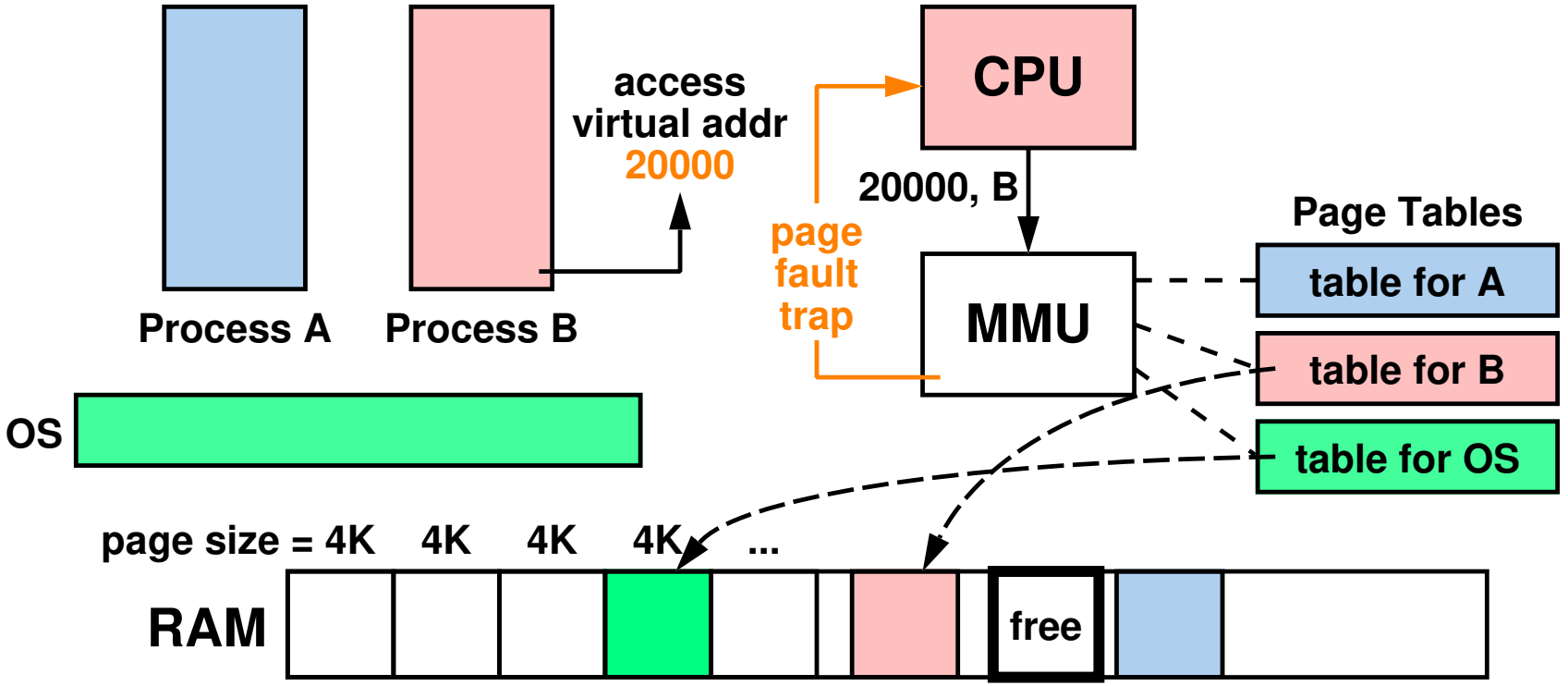


Page Table

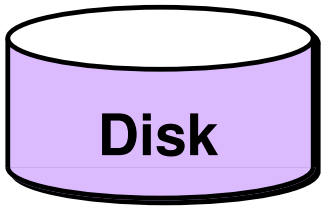


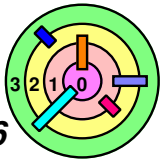
- ***Page fault***
  - page table does not have the requested address

Page Table

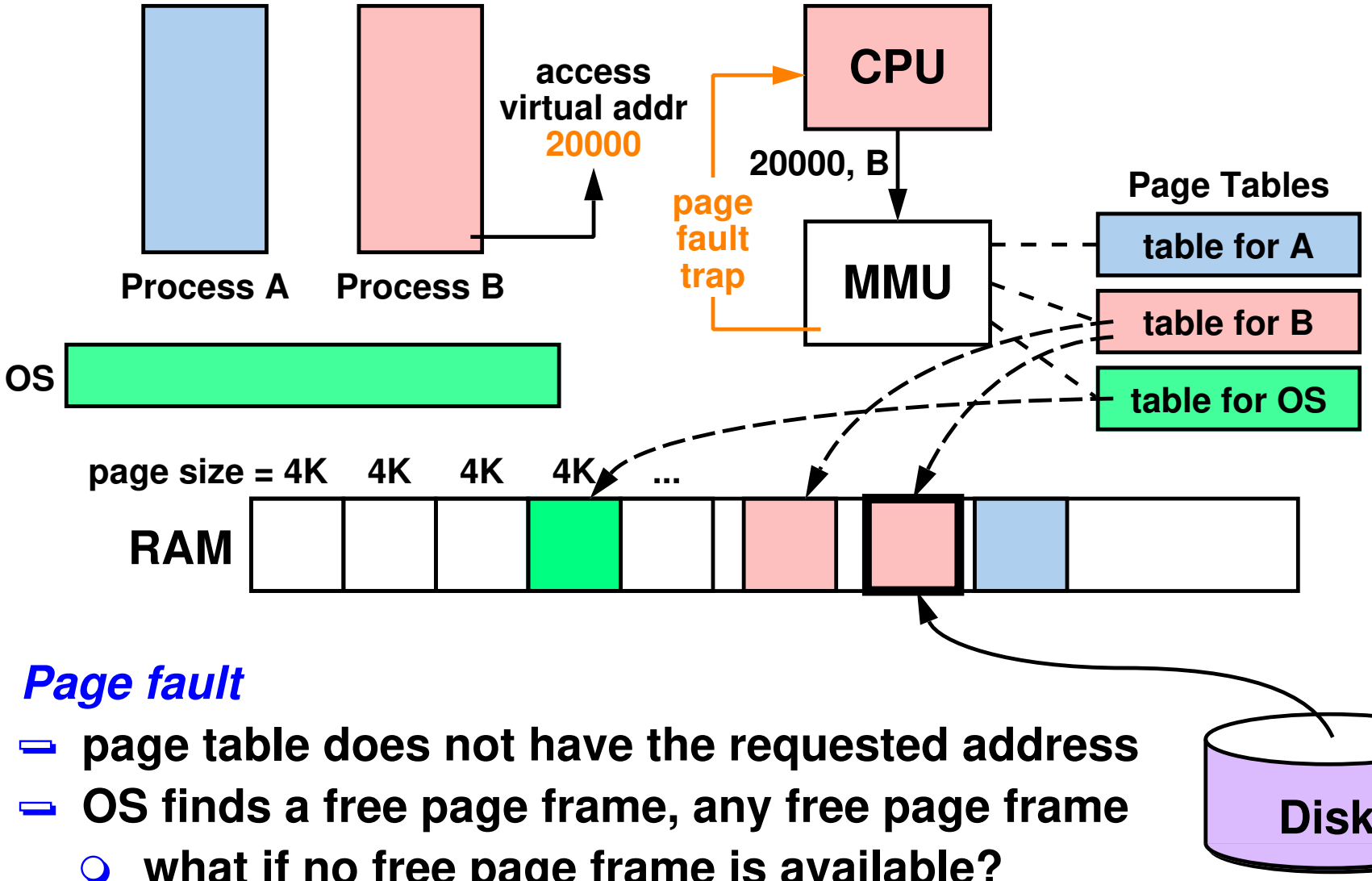


- ***Page fault***
  - page table does not have the requested address
  - OS finds a free page frame, any free page frame
    - what if no free page frame is available?

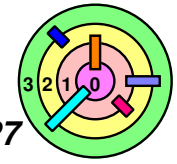




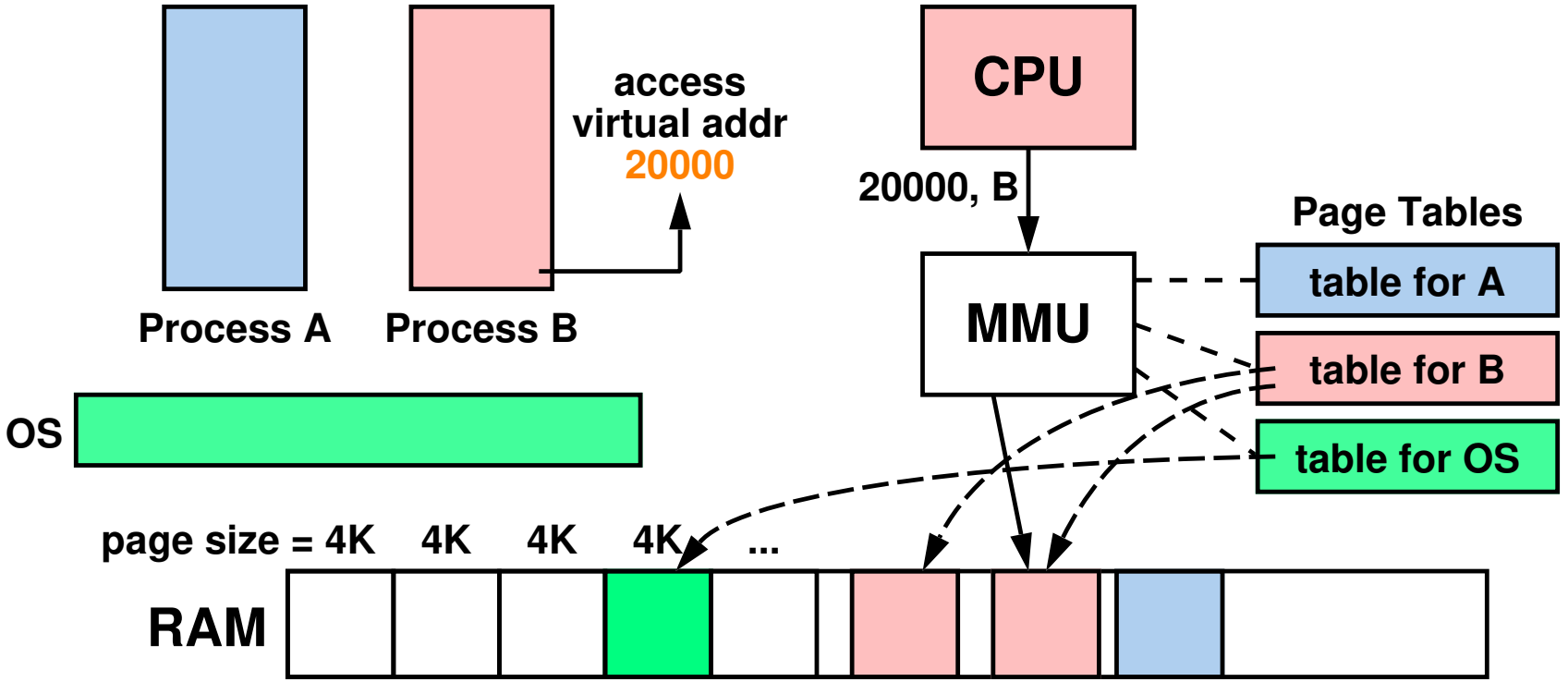
Page Table



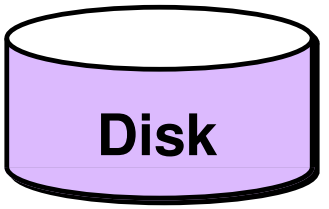
- ***Page fault***
  - page table does not have the requested address
  - OS finds a free page frame, any free page frame
    - what if no free page frame is available?
  - OS loads the requested page from disk

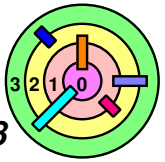


Page Table



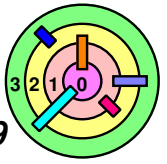
- ***Page fault***
  - page table does not have the requested address
  - OS finds a free page frame, any free page frame
    - what if no free page frame is available?
  - OS loads the requested page from disk
  - OS fixes page table and ***restarts*** user memory reference





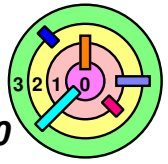
Basic (Two-level) Page Tables

- Two main problems
  1) performance
     - lookup in page table requires one memory access
     - to access memory starting with a virtual address will take at least *two* memory accesses
       - that's twice as slow
  2) page table takes up a lot of space
     - most entries are not used
- We will first look at solutions for the space problem

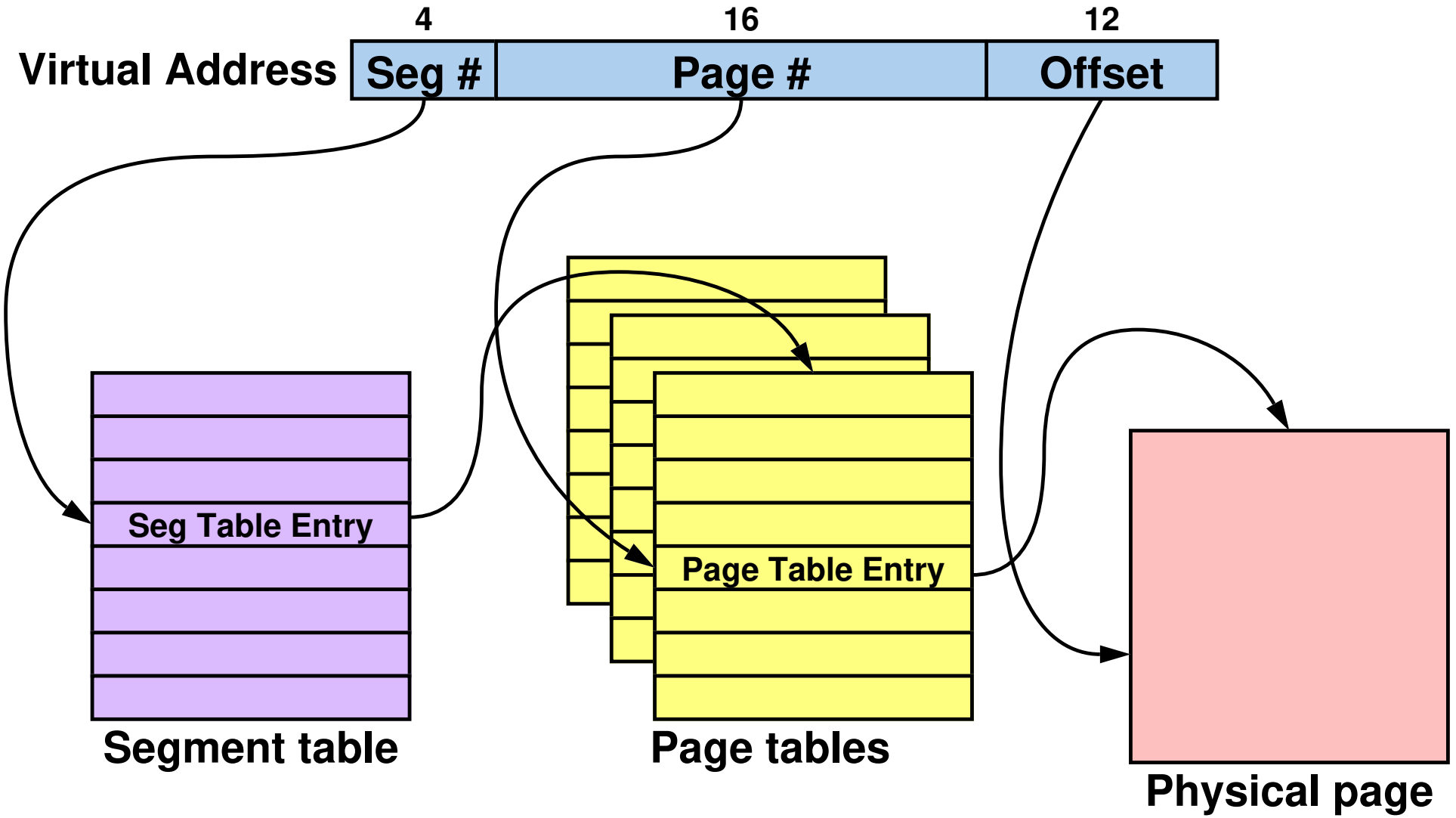


Page-Table Size

- Consider a full 2^{32}-byte address space
  - assume 4096-byte (2^{12}-byte) pages
  - 4 bytes per page table entry
  - the page table would consist of $2^{32}/2^{12}$ (= 2^{20}) entries
  - its size would be 2^{22} bytes (or 4 megabytes)
- This is a general *scaling* problem
  - solutions:
    - hierarchy *(e.g., forward-mapped page tables)* or *hash page tables*
    - *virtual linear page tables* (i.e., page tables in virtual memory)

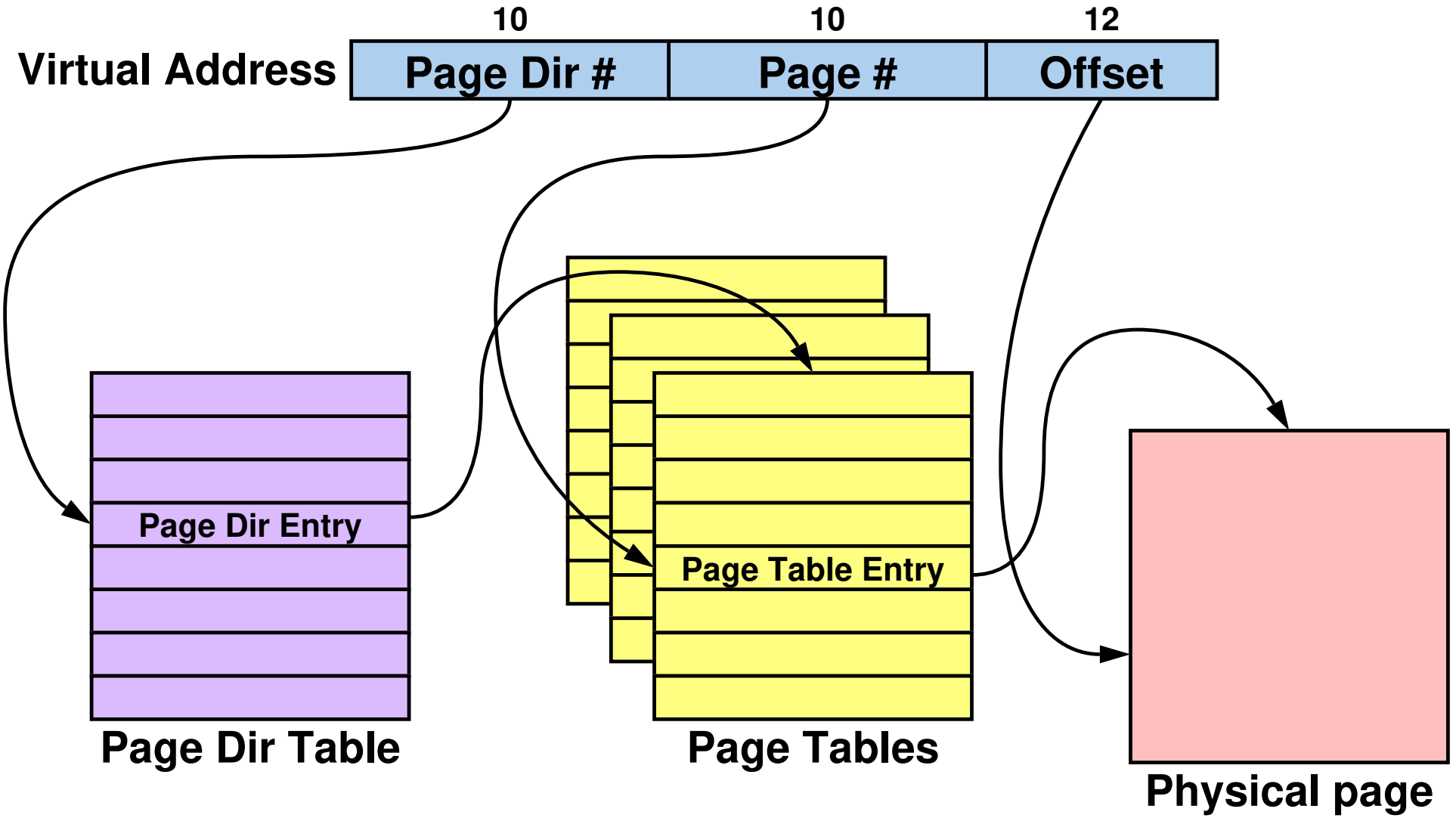


Paged Segmentation



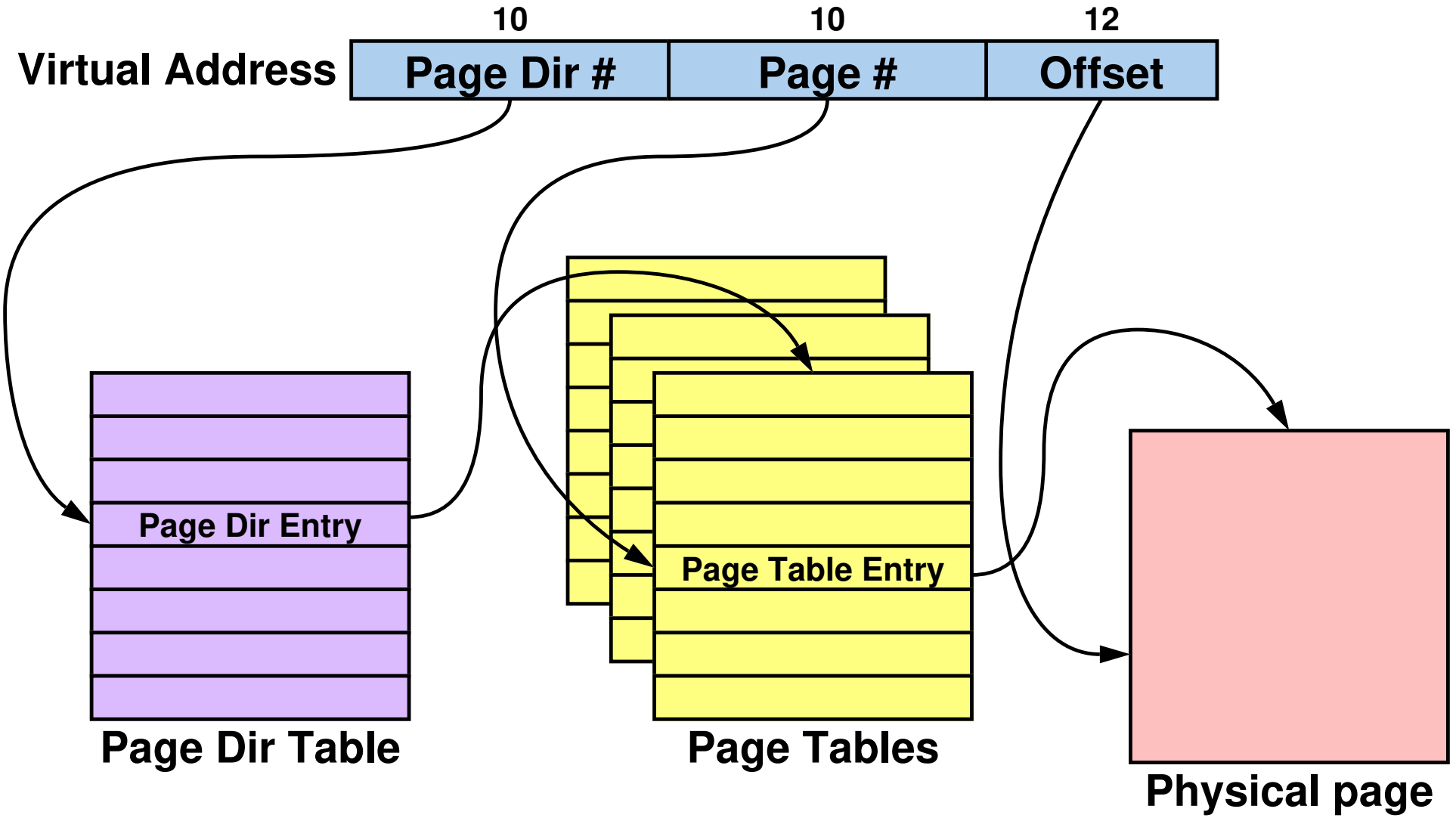
- If a process only uses 4 segments, it would need 4 page tables
  - a total of 256K page table entries (if an entry is 4 bytes long, this would take up 1MB)

Forward-Mapped (Multilevel) Page Table



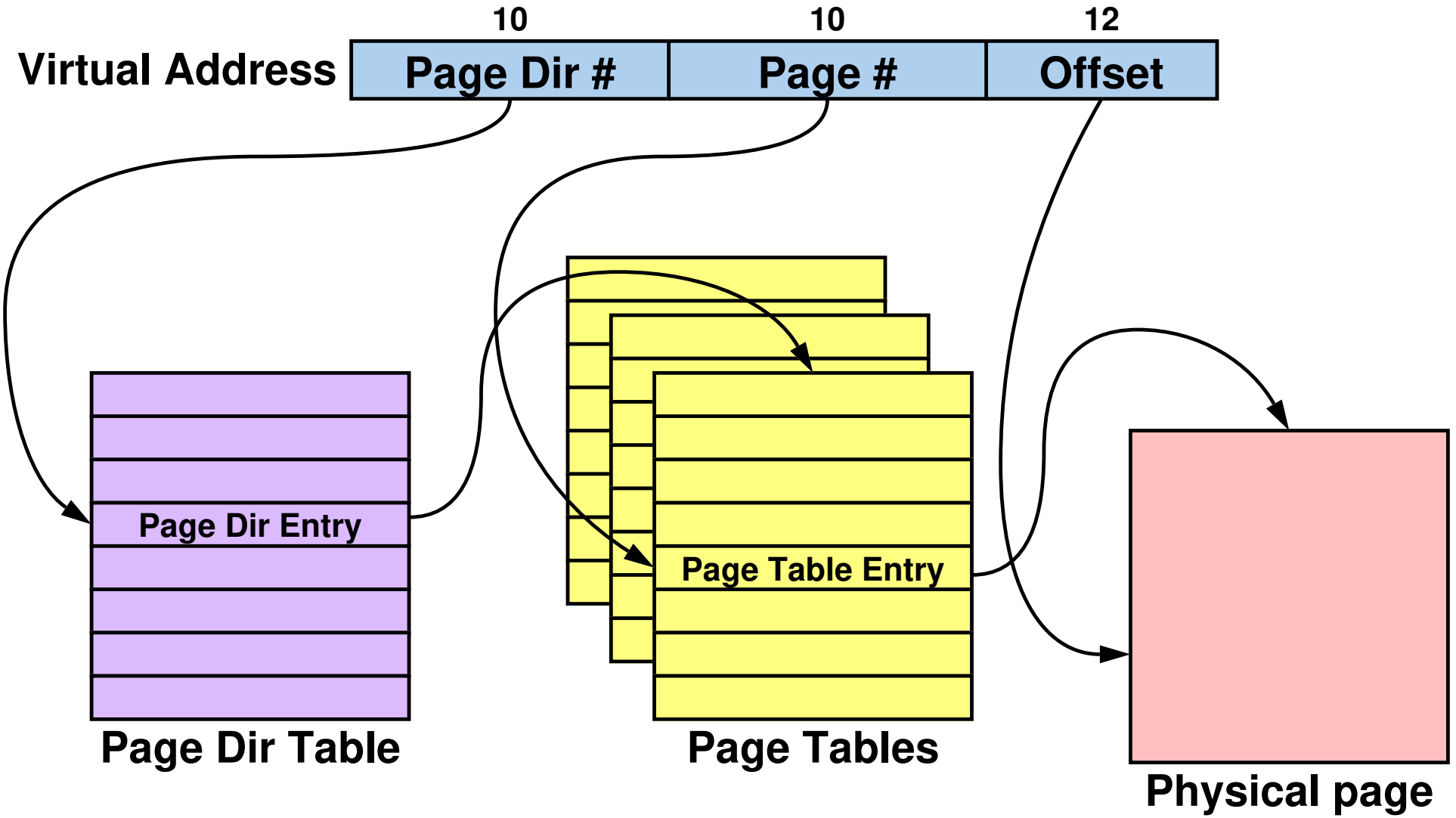
- Intel x86

Forward-Mapped (Multilevel) Page Table



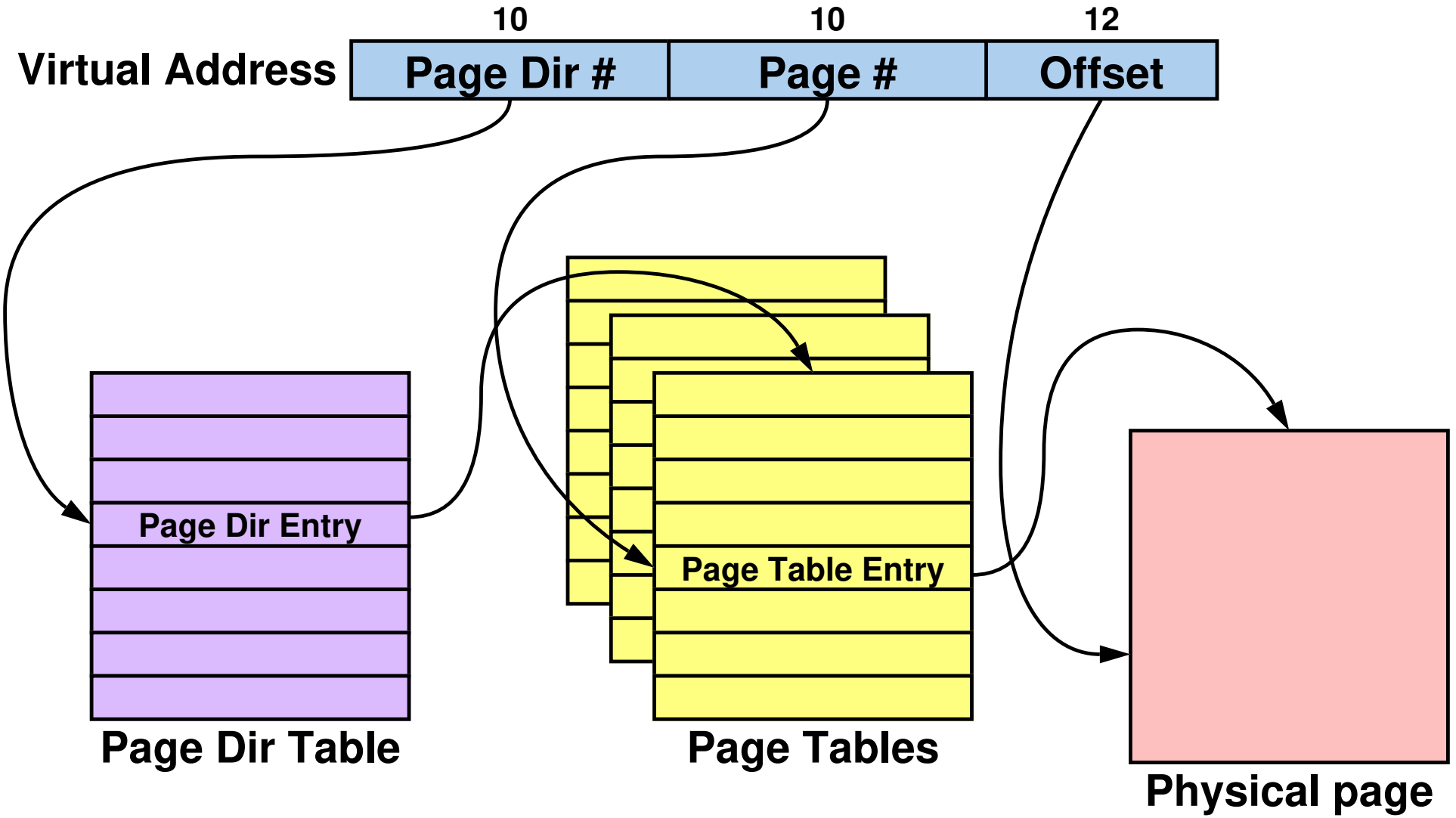
What's the minimum page table overhead?

Forward-Mapped (Multilevel) Page Table



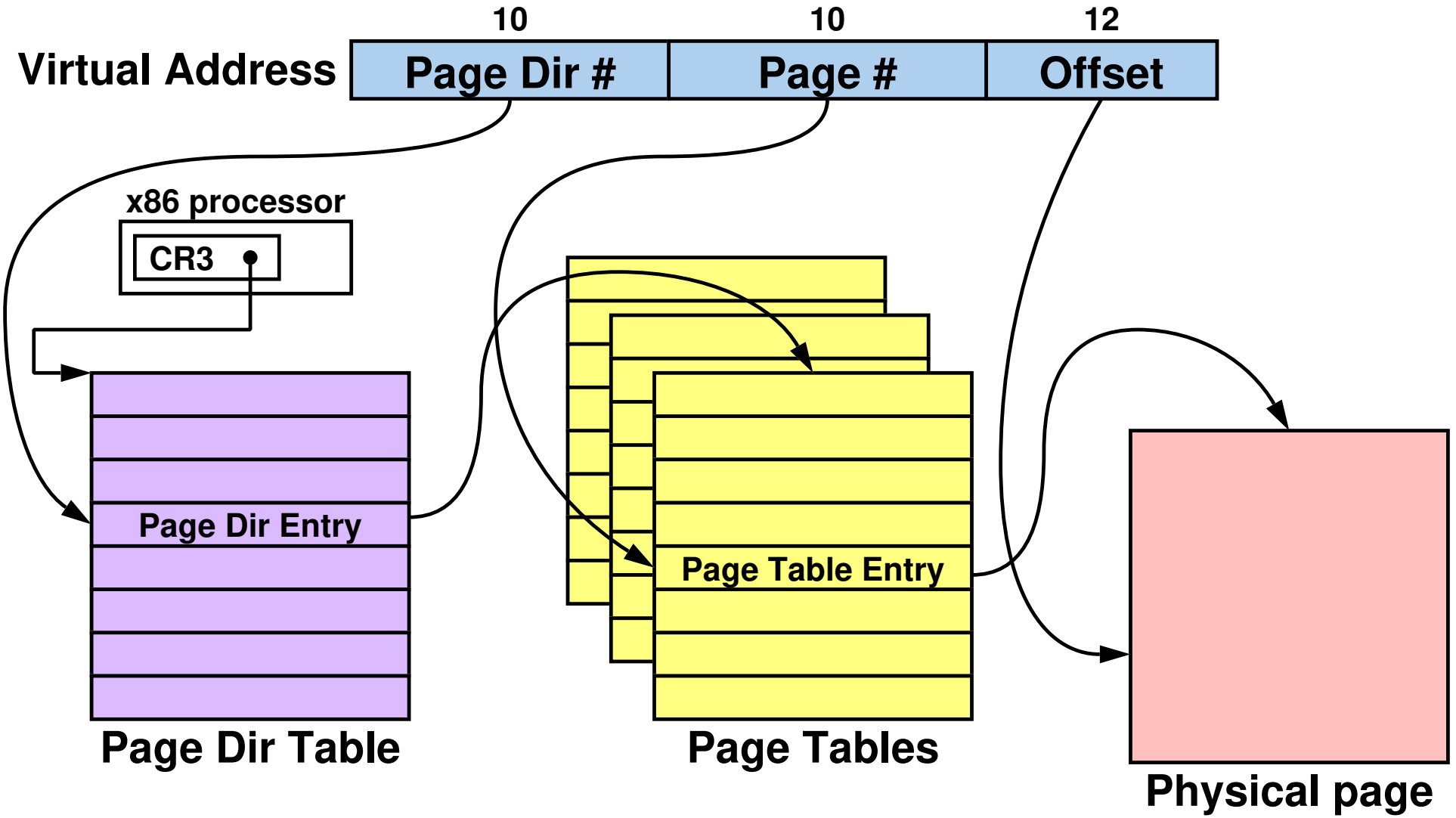
- What's the minimum page table overhead?
  - 16KB - one page dir table and three page tables (two for low addresses and one for high addresses)

Forward-Mapped (Multilevel) Page Table



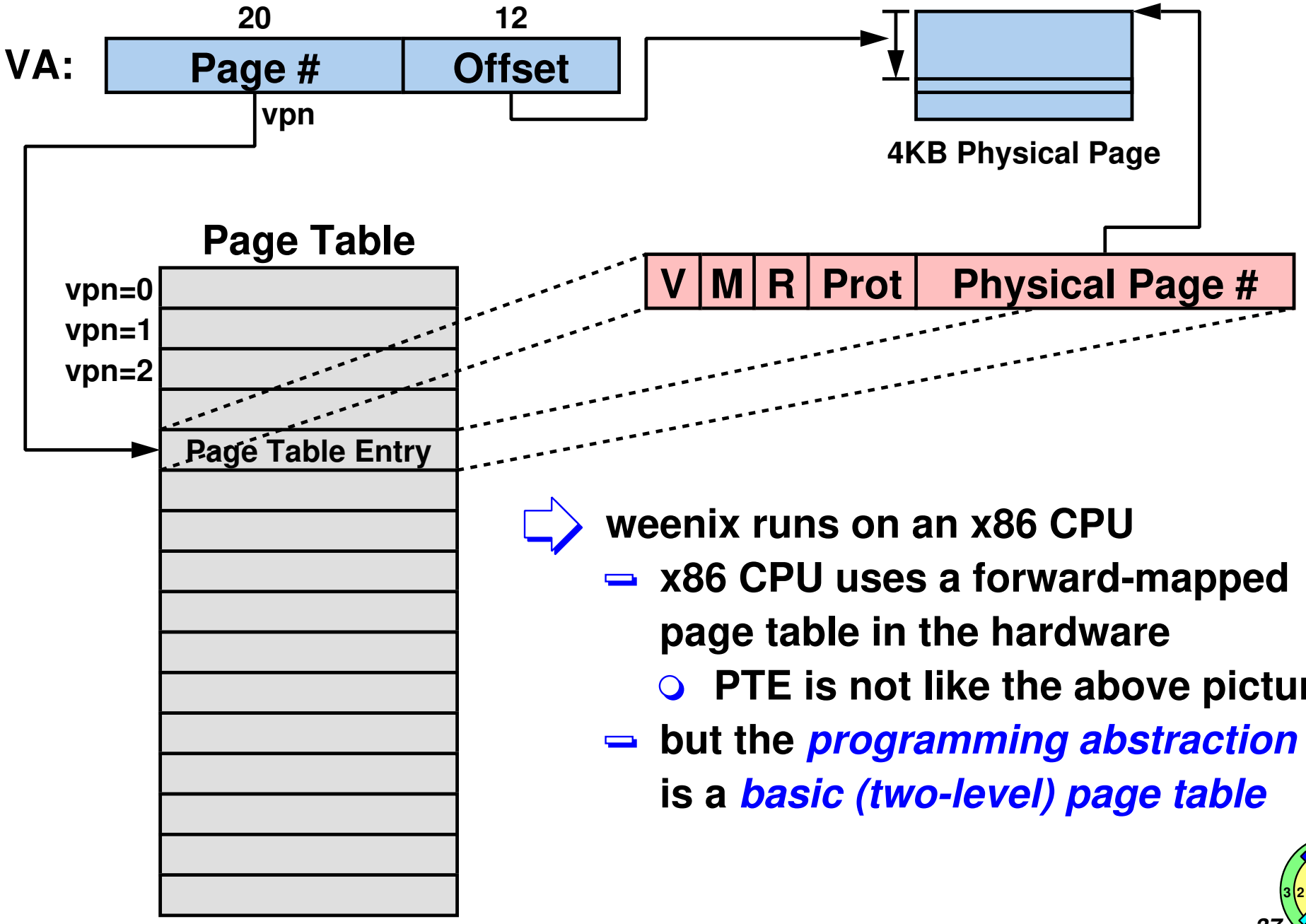
- Main drawback (will deal with this later)
  - *two* physical memory accesses just to *map* a virtual address to a physical address

Forward-Mapped (Multilevel) Page Table



- Forward-mapped page table is used in x86 processors
  - CR3 contains ***physical address*** of the base address of the page directory table of the currently running process

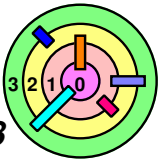
Forward-Mapped (Multilevel) Page Table



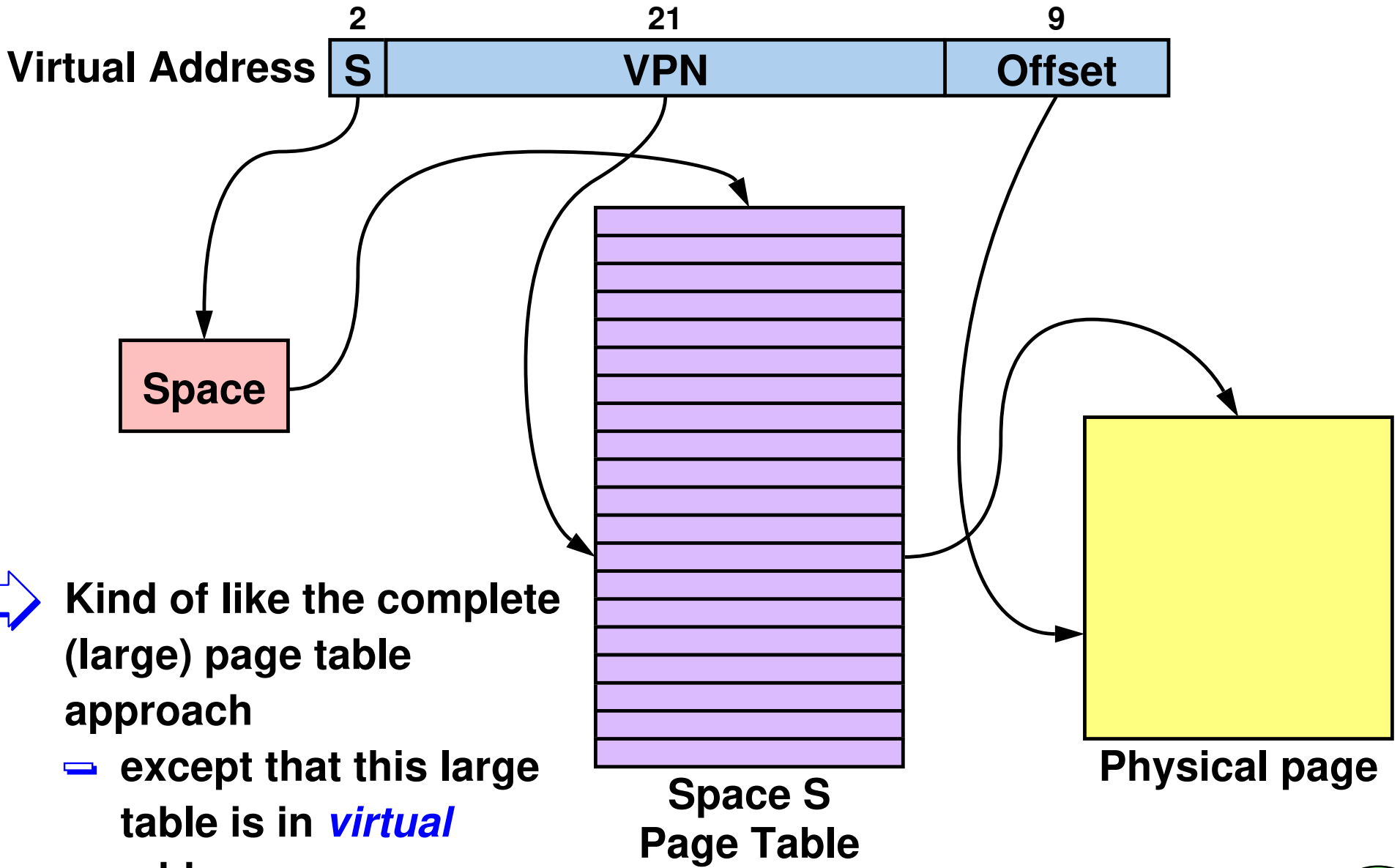
- weenix runs on an x86 CPU
  - x86 CPU uses a forward-mapped page table in the hardware
    - PTE is not like the above picture
  - but the *programming abstraction* is a *basic (two-level) page table*

7.2 Hardware Support for Virtual Memory

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

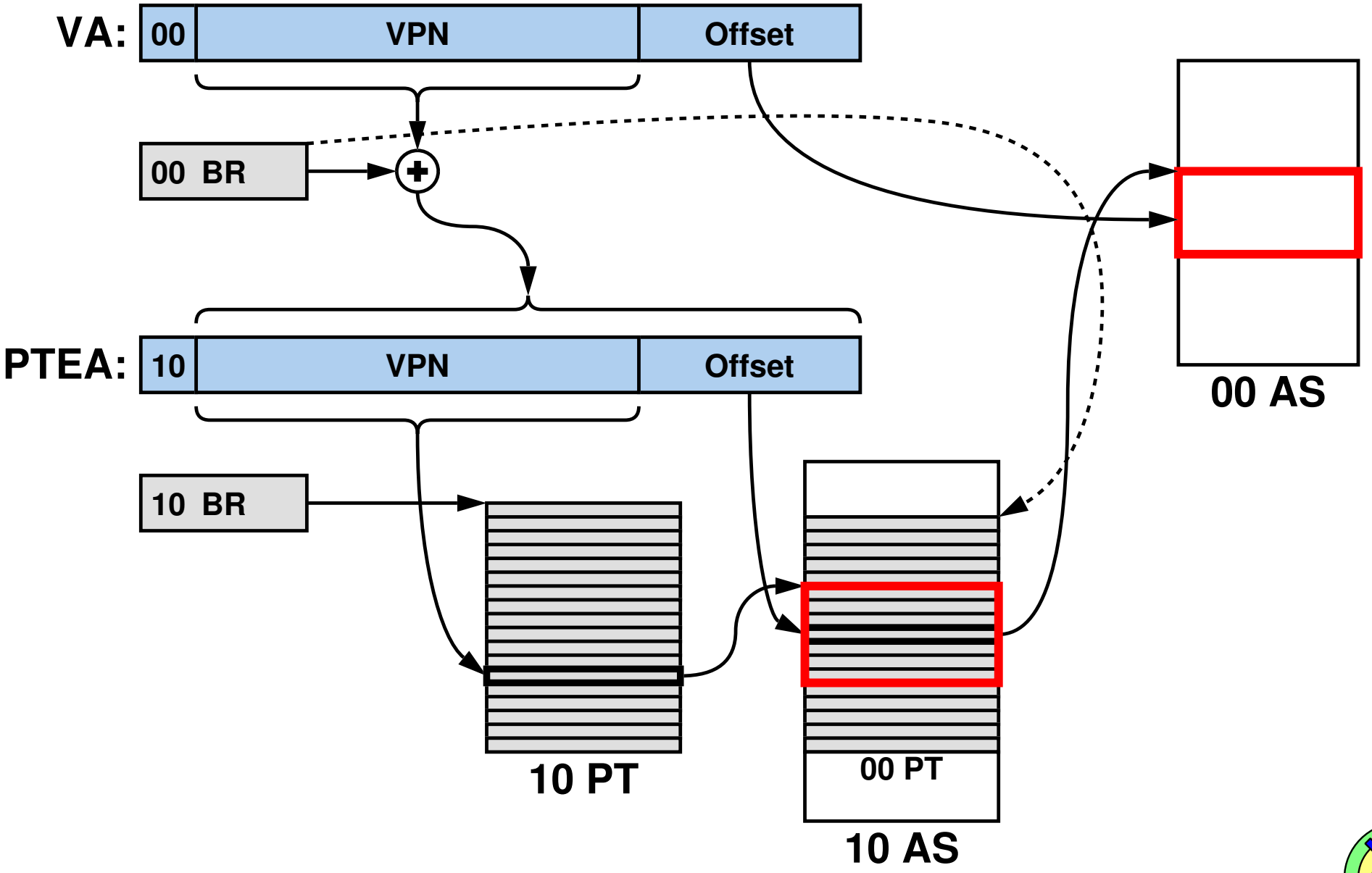


Linear Page Table



- Kind of like the complete (large) page table approach
  - except that this large table is in *virtual* address space

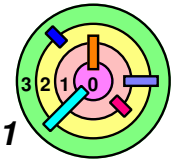
VAX Linear Page Translation



PTEA: Page Table Entry Address

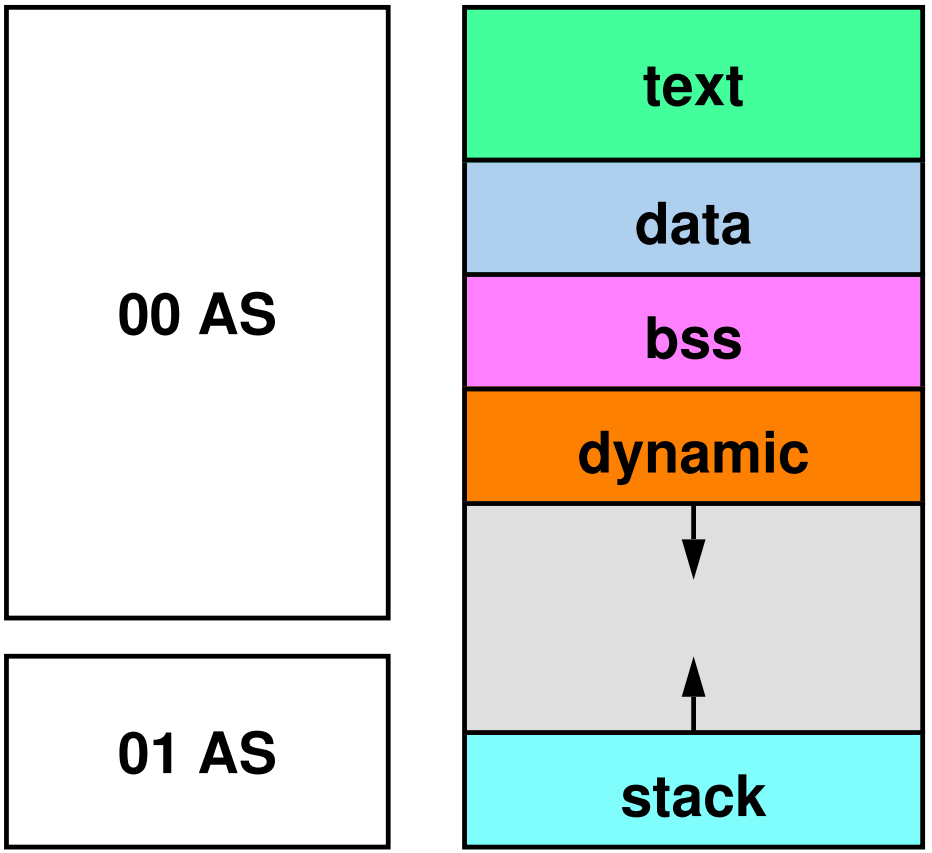
Linear Page Table Management

- 00 and 01 page tables each require contiguous locations in 10 space
  - with 512-byte pages, 8MB (= 4bytes $\times 2^{21}$) each:
    - memory cost was $40,000 per MB for the VAX
    - maximum of 64 such page tables (to fill up 500MB of physical memory)
    - (need room for other things, e.g. OS)
- Reduce size requirements with partial page tables
  - *length register* constrains size of each space



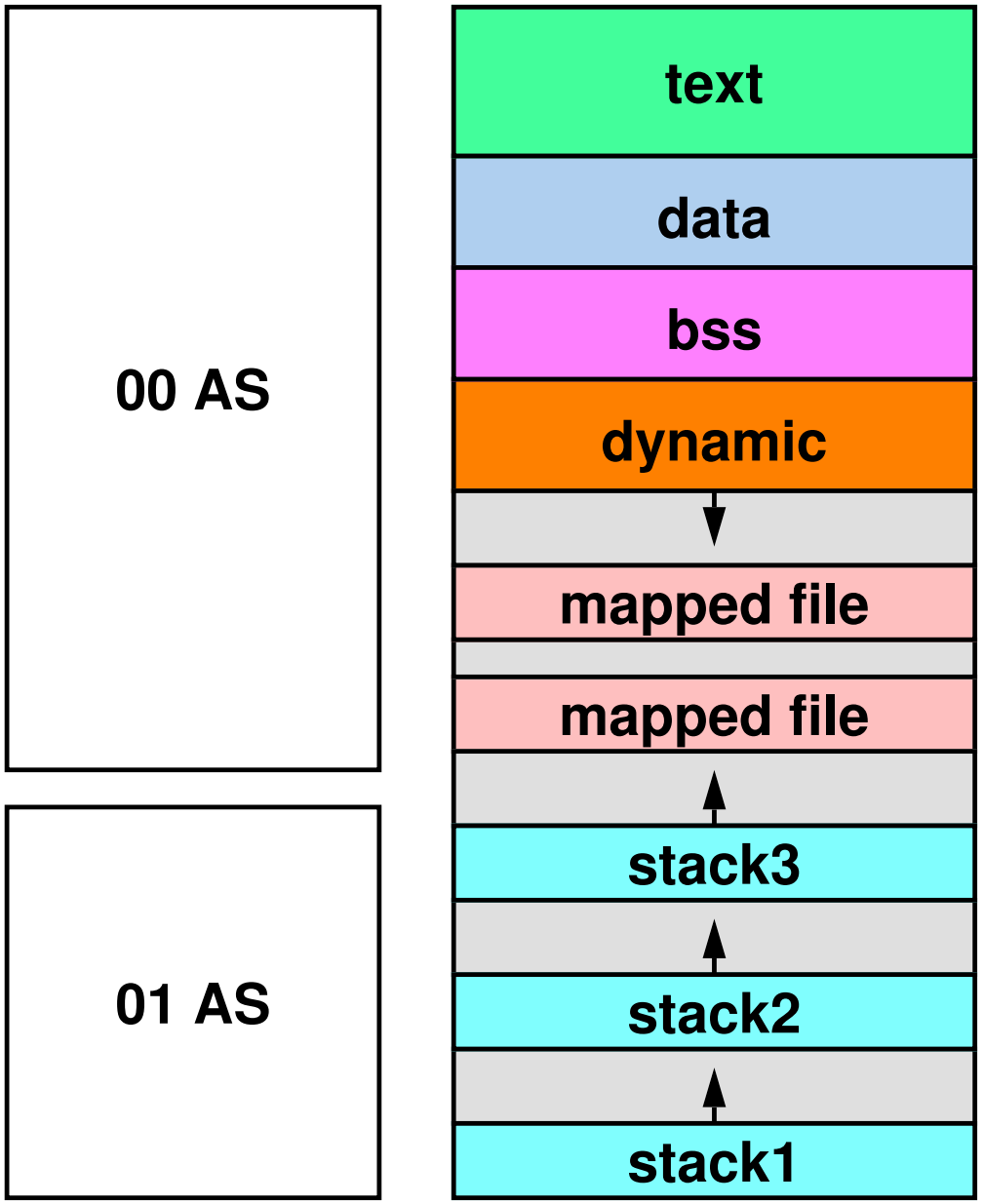
Traditional Unix with Linear PTs

- With traditional Unix, using a length register worked pretty well



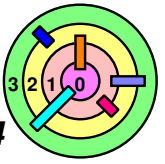
Modern Unix

- Not so well with modern Unix



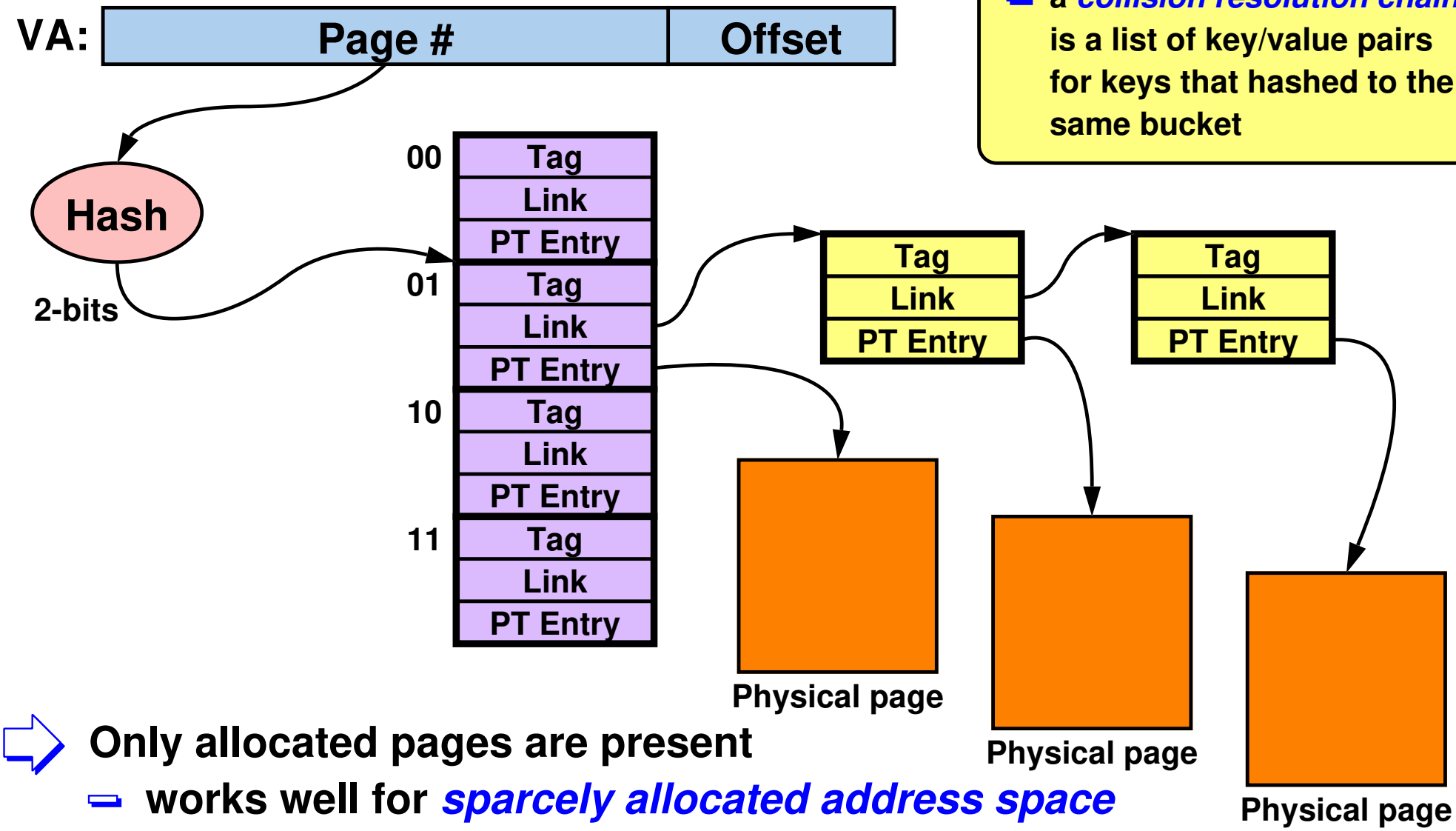
7.2 Hardware Support for Virtual Memory

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



Hashed Page Tables

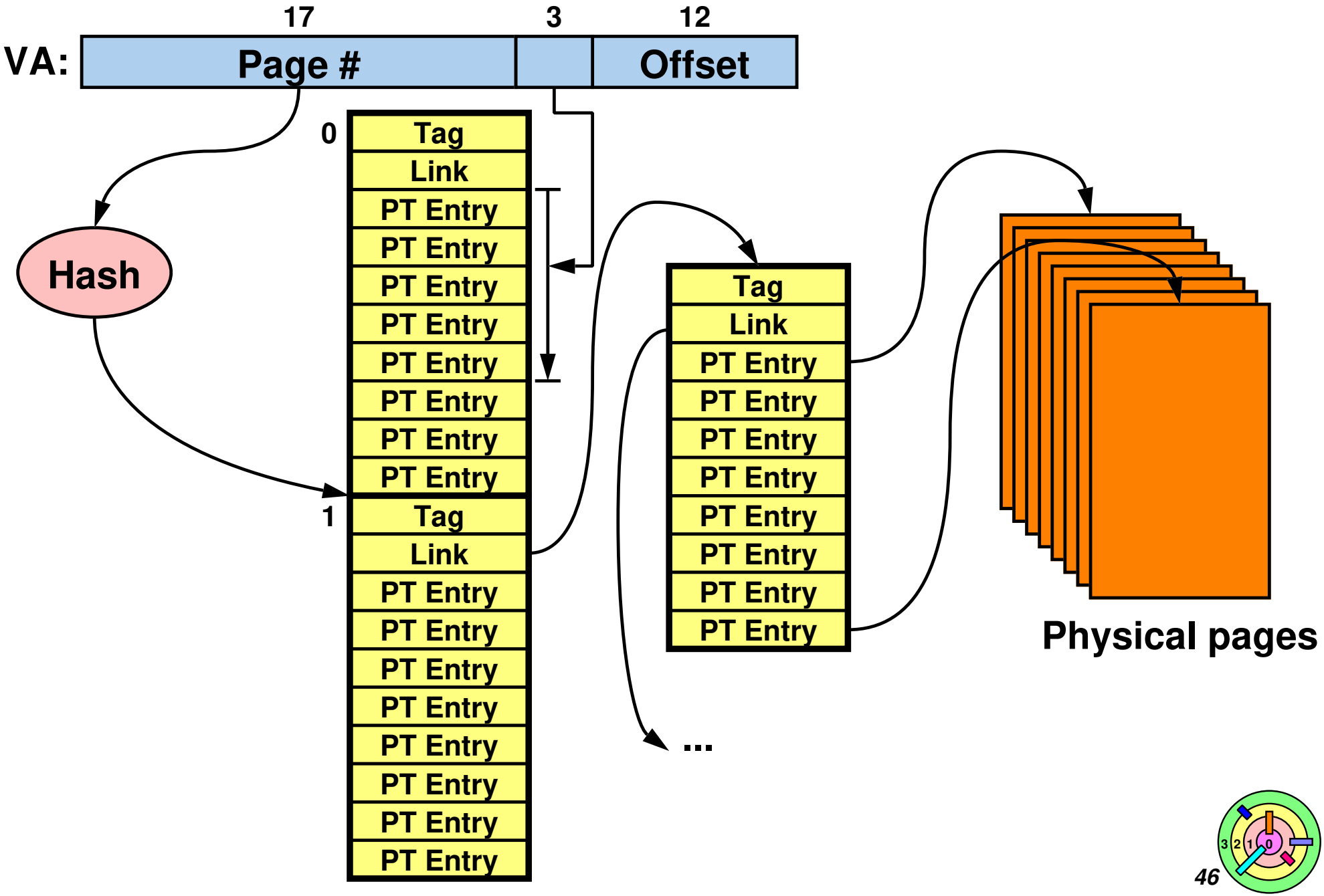
- a *collision resolution chain* is a list of key/value pairs for keys that hashed to the same bucket



⇨ Only allocated pages are present

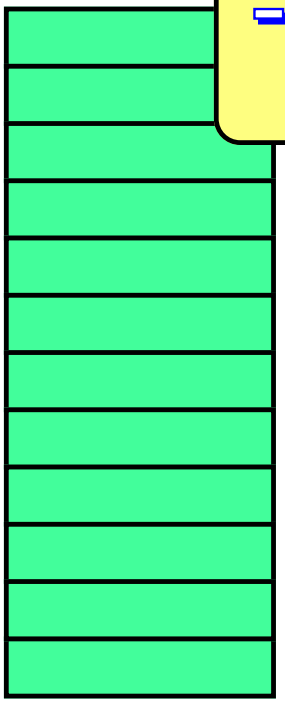
- works well for *sparcely allocated address space*
- too much overhead (tag and link) when there are large regions of allocated memory

Clustered Page Tables



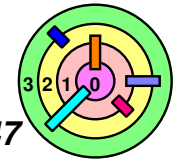
Inverted Page Tables

Each entry contains
- PID
- page number
- physical page number

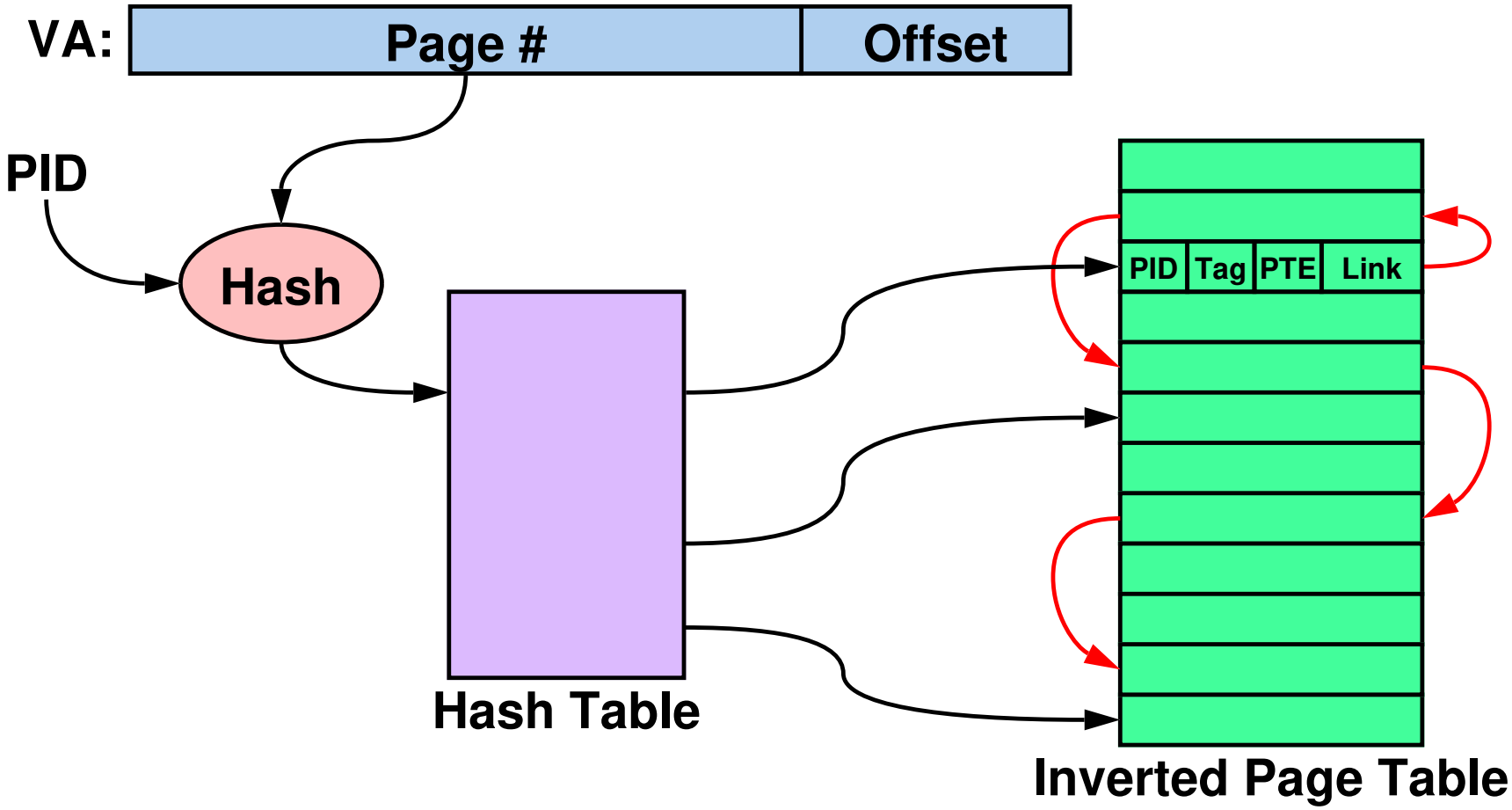


Inverted Page Table

- Page table is *indexed* by *physical page number*
  - number of entries in IPT tends to be limited and small
  - how do you map page number (i.e., VPN) to physical page number?



Inverted Page Tables



- Page table is *indexed* by *physical page number*
  - number of entries in IPT tends to be limited and small
  - how do you map page number (i.e., VPN) to physical page number?