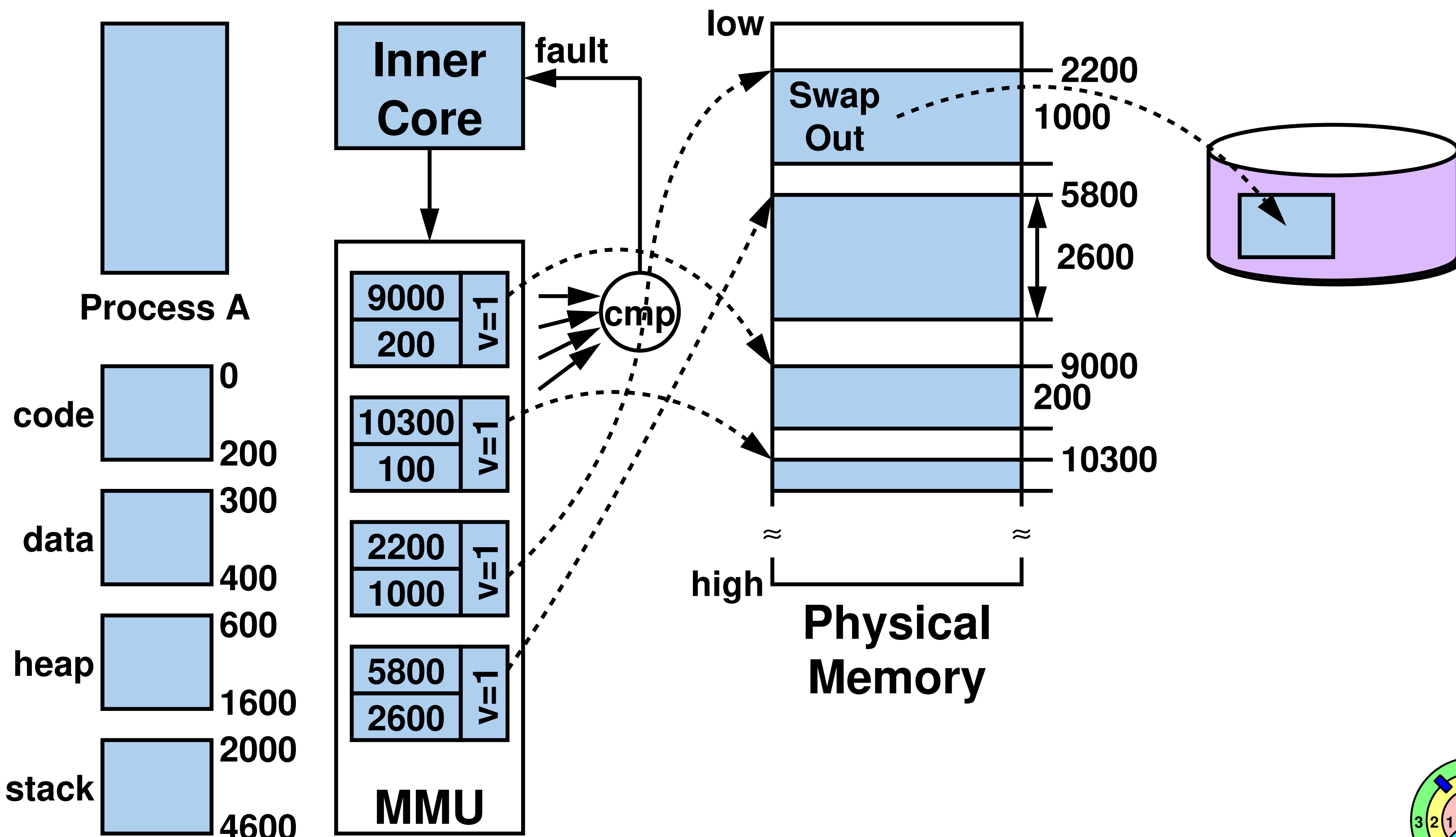
Swapping / Backing Store

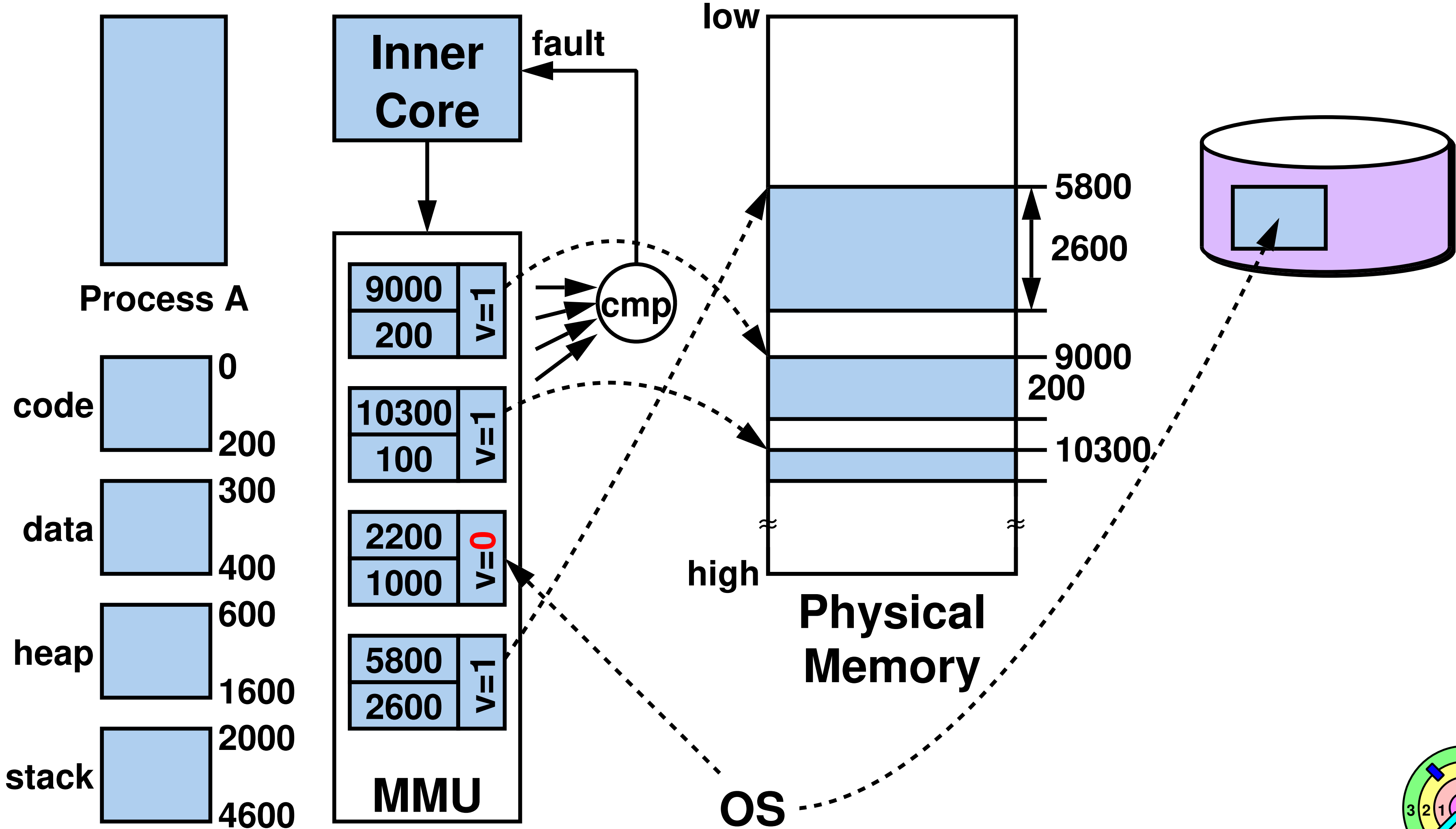
- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits, which are omitted from the picture below)



Copyright © William C. Cheng

Swapping / Backing Store

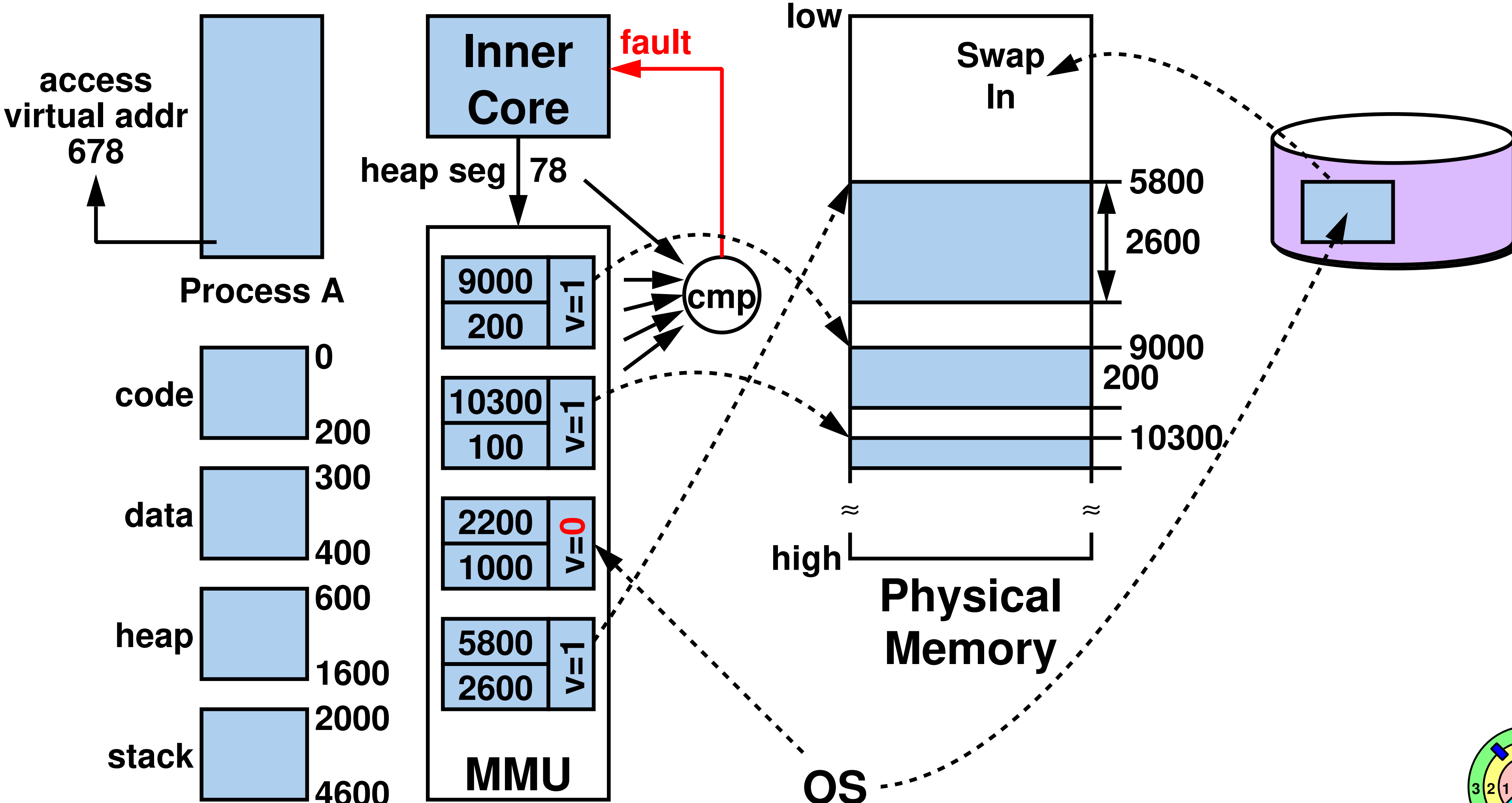
- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits, which are omitted from the picture below)



Copyright © William C. Cheng

Swapping / Backing Store

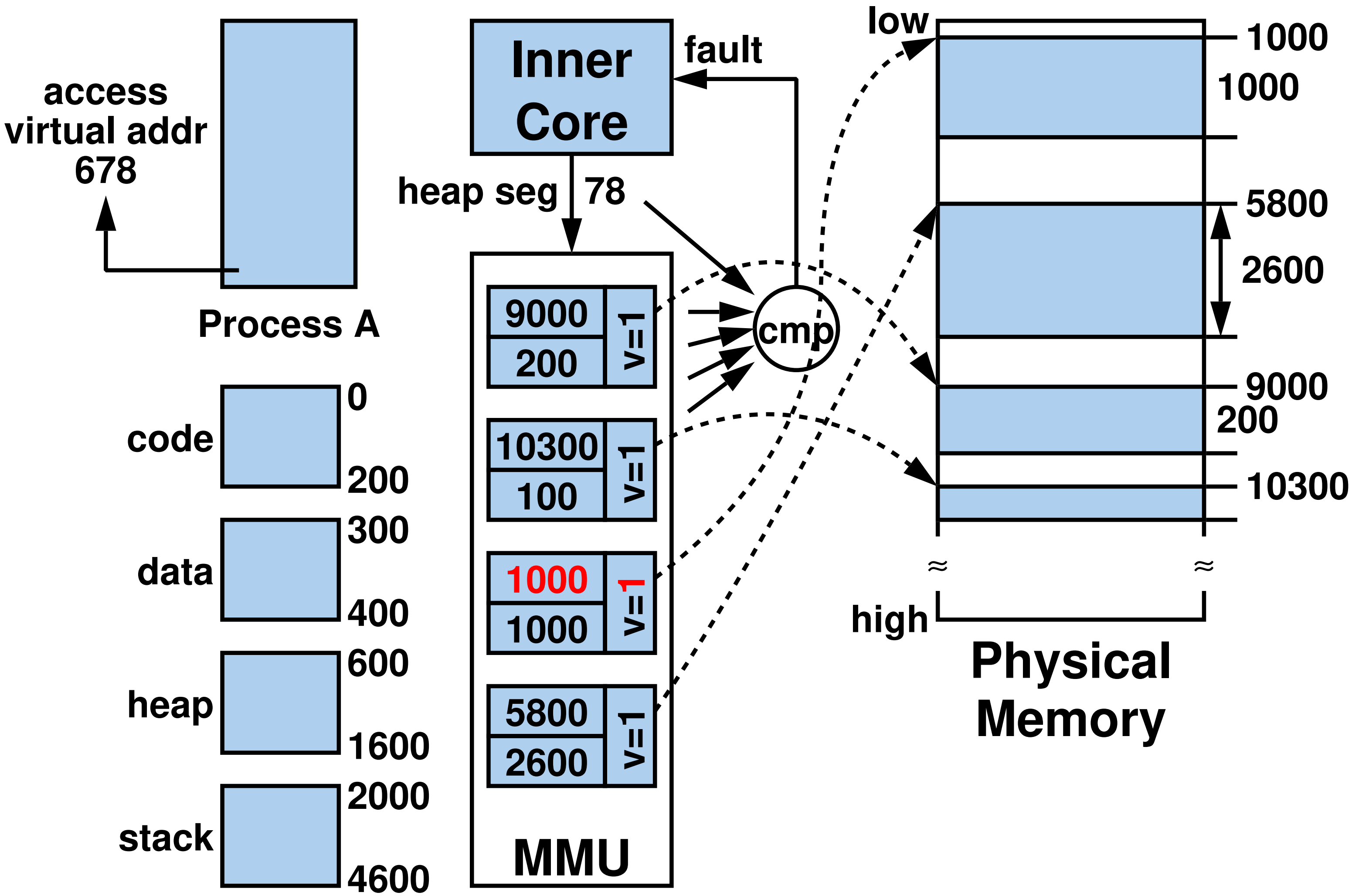
- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits, which are omitted from the picture below)



Copyright © William C. Cheng

Swapping / Backing Store

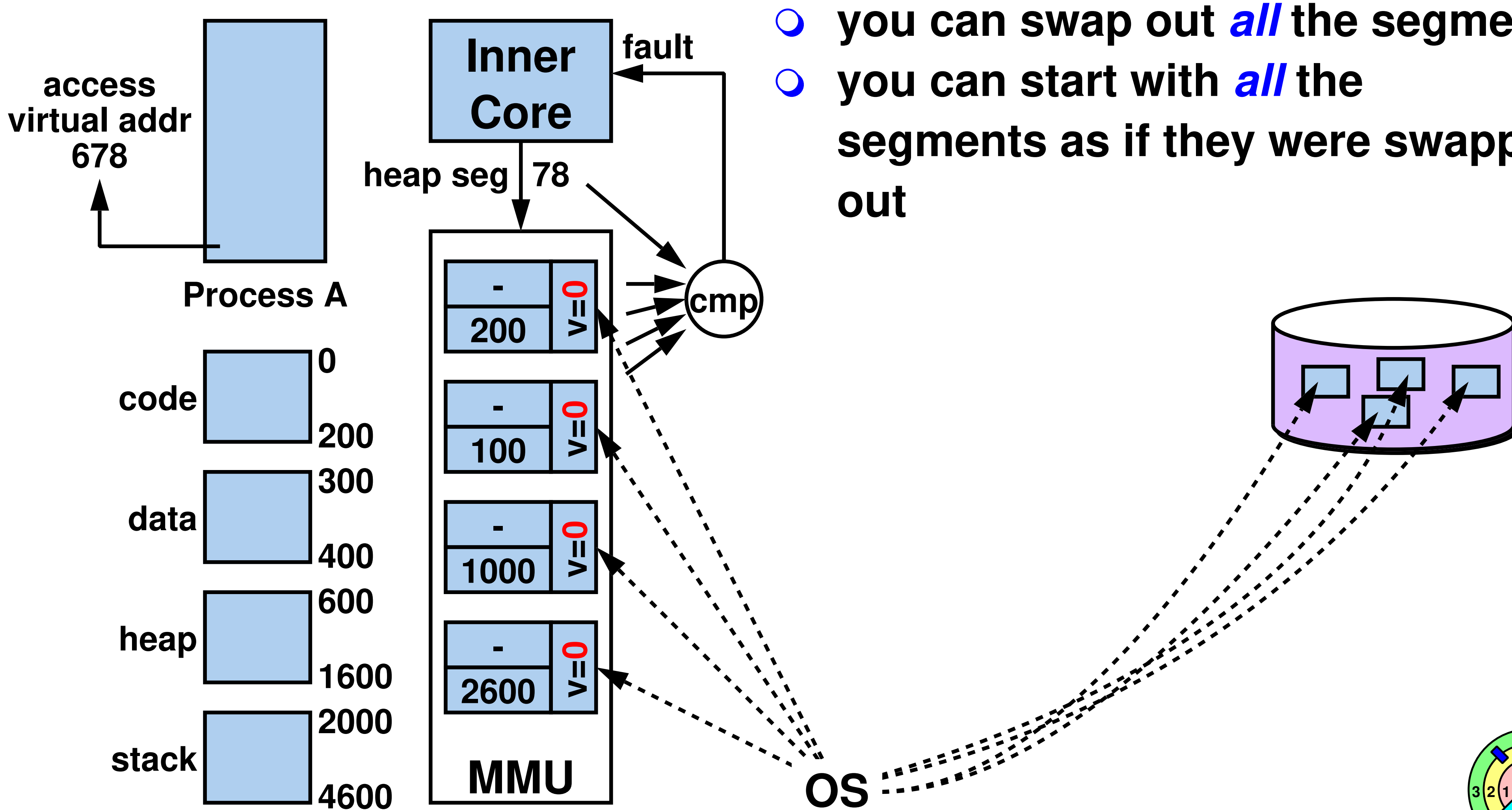
- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits, which are omitted from the picture below)



Copyright © William C. Cheng

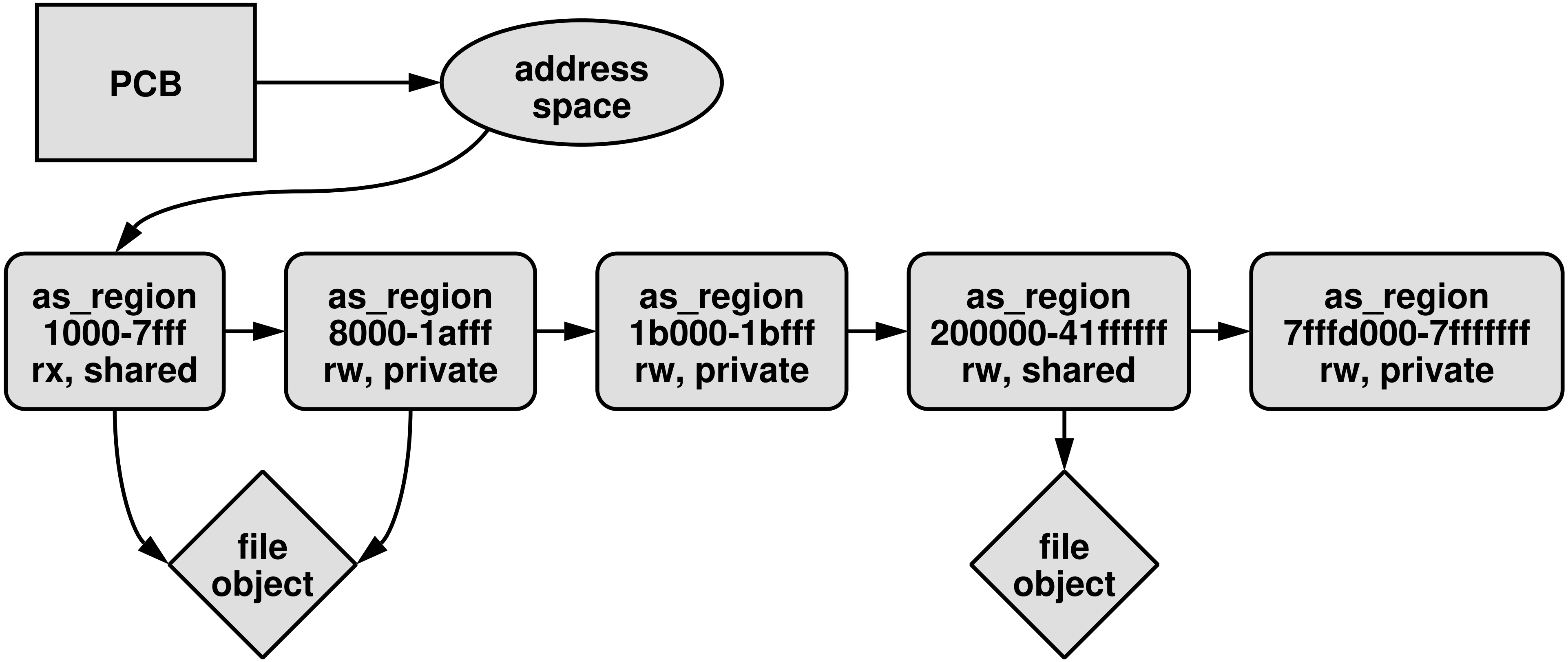
Swapping / Backing Store

- 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, which are omitted from the picture below)
    - you can swap out *all* the segments
    - you can start with *all* the segments as if they were swapped out



Copyright © William C. Cheng

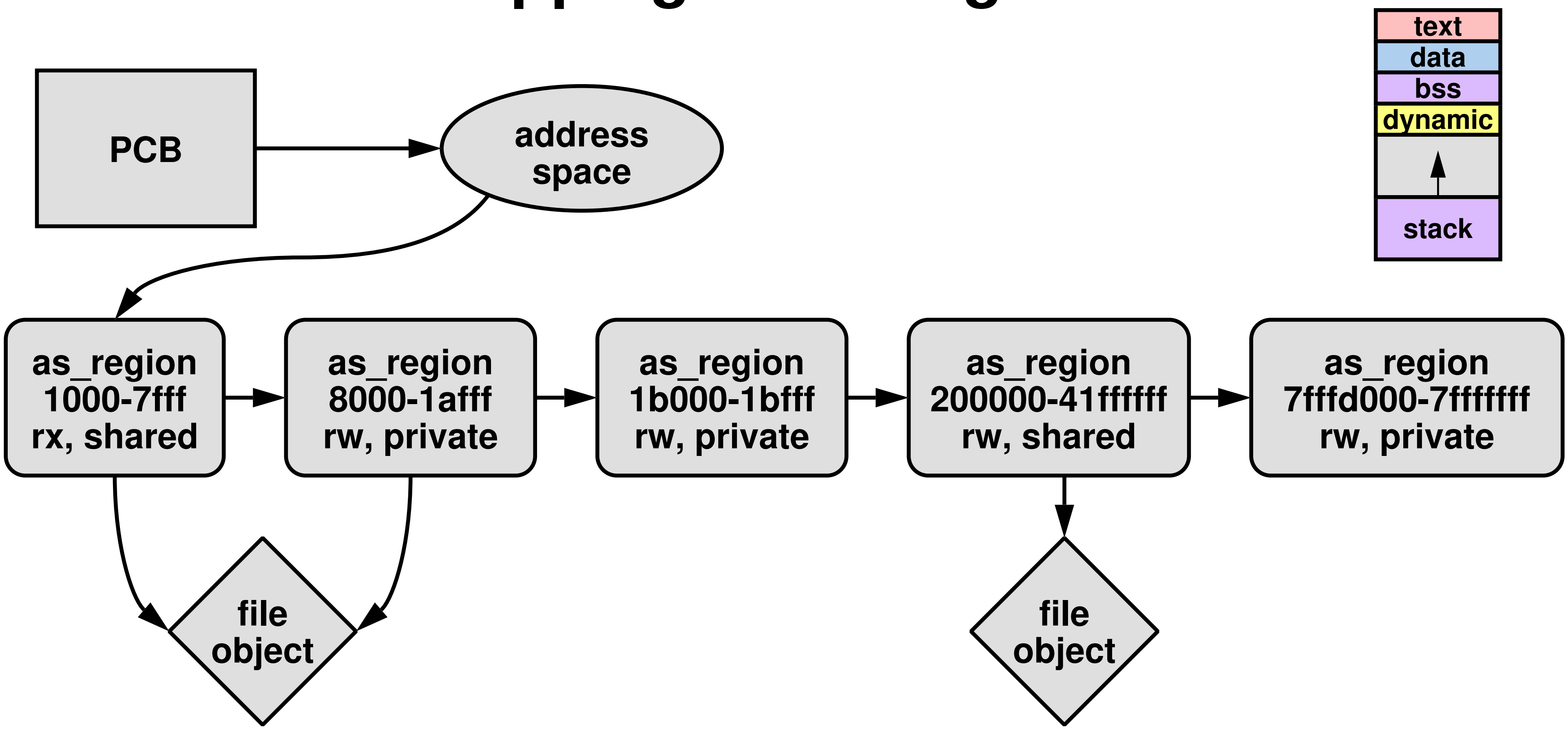
Swapping / Backing Store



Remember this?

Copyright © William C. Cheng

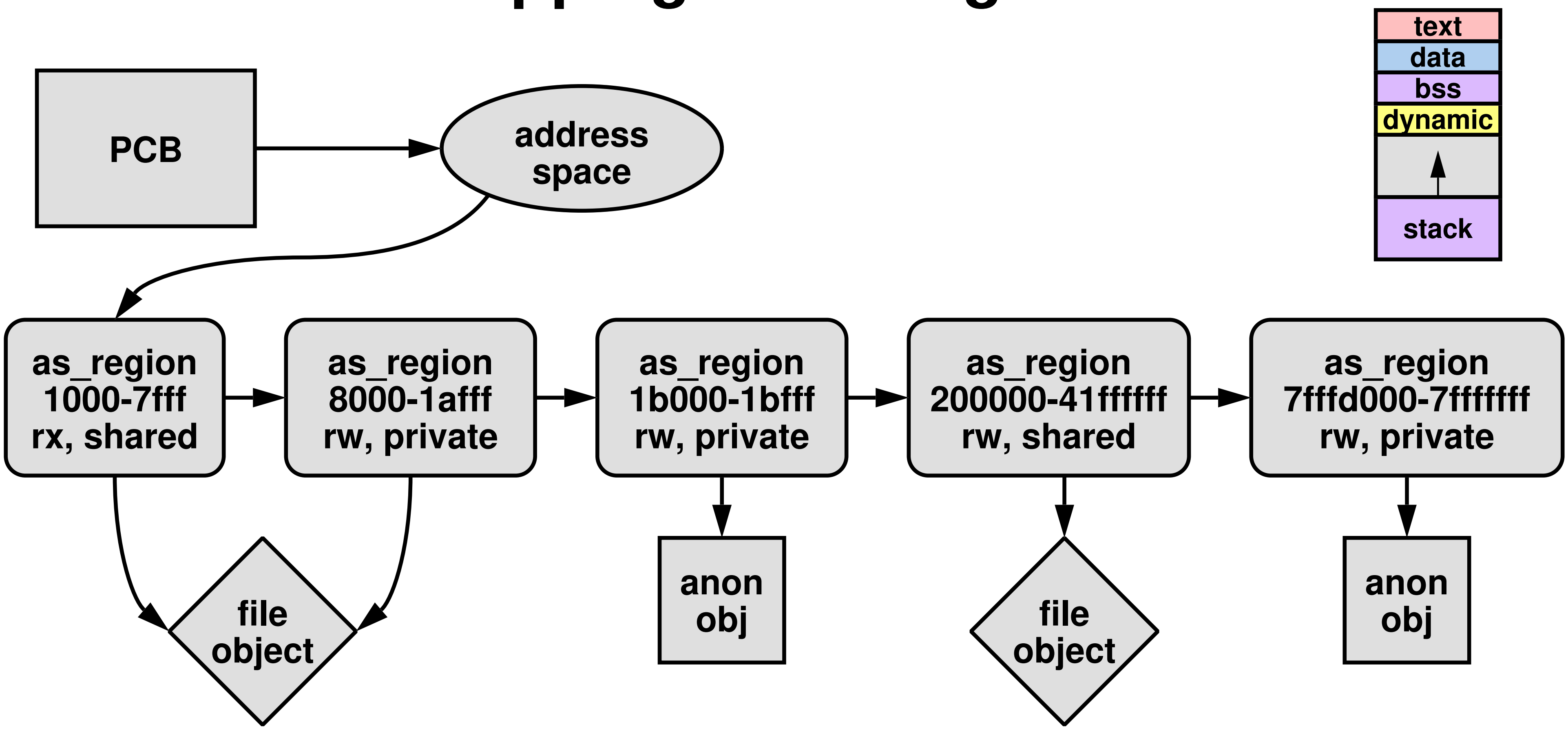
Swapping / Backing Store



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

Copyright © William C. Cheng

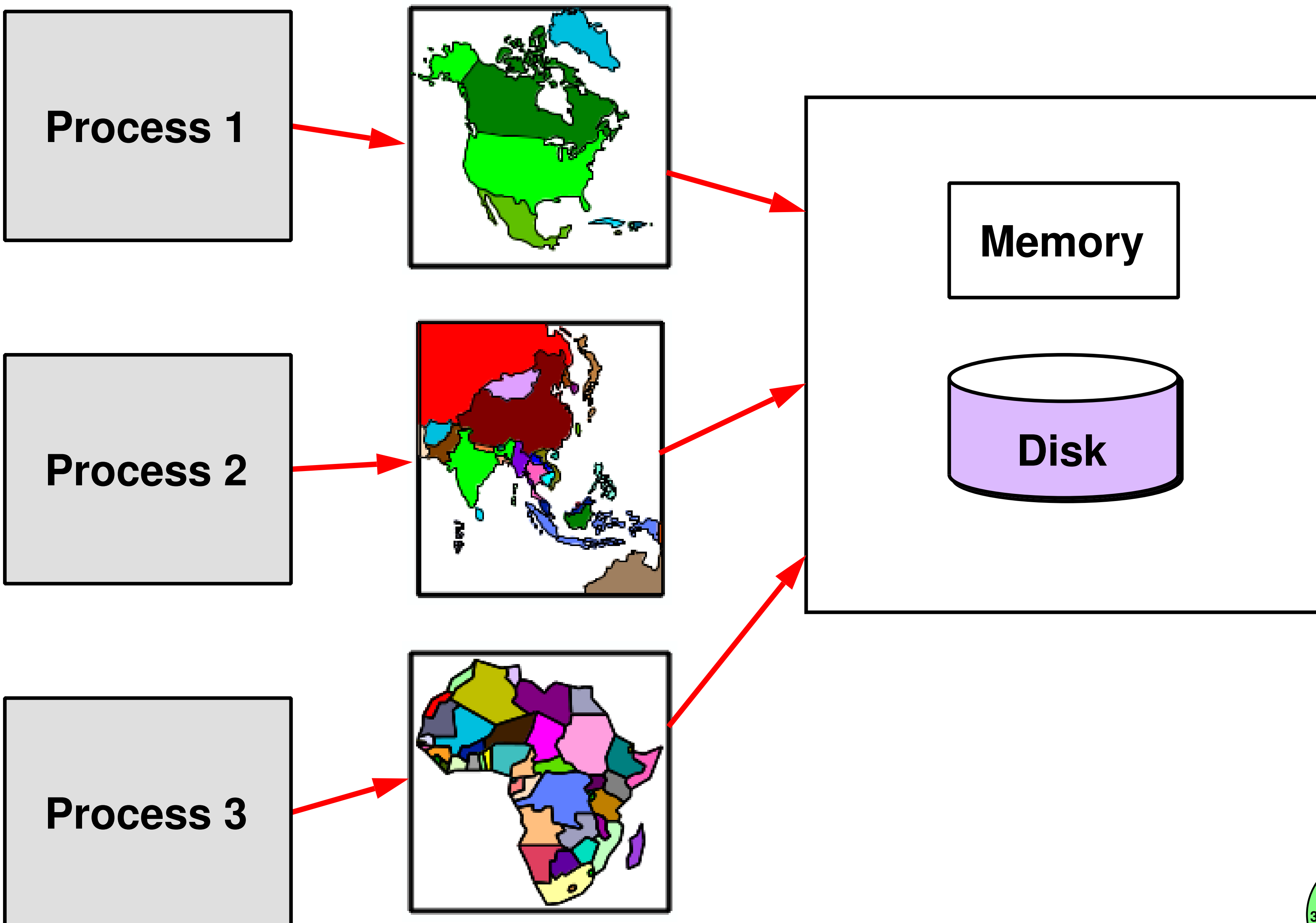
Swapping / Backing Store



- Every *user space memory segment* needs a corresponding disk image, in case the user process needs to be *swapped out*
  - what about *kernel* memory?
    - it's OS-dependent - in Windows, *some* kernel memory can be swapped out

Copyright © William C. Cheng

Virtual Memory



Copyright © William C. Cheng

7.2 Hardware Support for Virtual Memory

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

Copyright © William C. Cheng

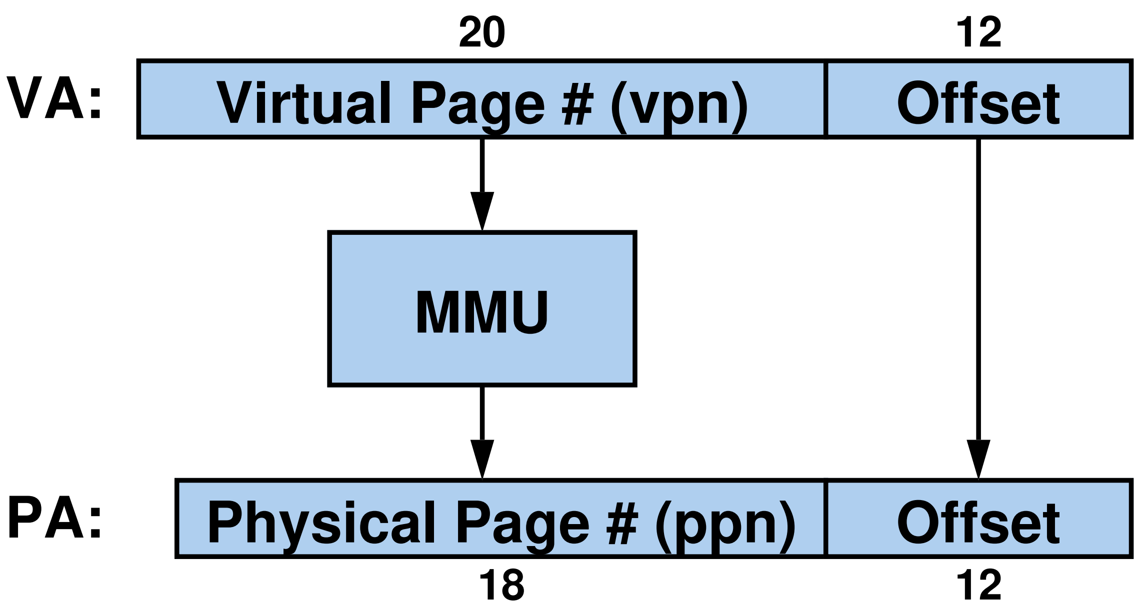
Structuring Virtual Memory

- ***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**

Copyright © William C. Cheng

Paging

- Map a 32-bit virtual address to physical address (number of bits depends on how much physical memoroy you have)
  - think of all memory as an *array of fixed size pages*
  - we will assume page size is 4KB = 2^{12}
- Ex: 1GB (2^{30}) of physical memory (divide into pages of same size)



- Address translation in paging: convert *virtual page number* to *physical page number*
  - offset within page stays the same

Copyright © William C. Cheng

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 *lookup table* to map *virtual page numbers* to *physical page numbers*
- Ex: 1GB of physical memory with 4KB pages
  - 2^{18} physical pages
  - terminology: 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)*

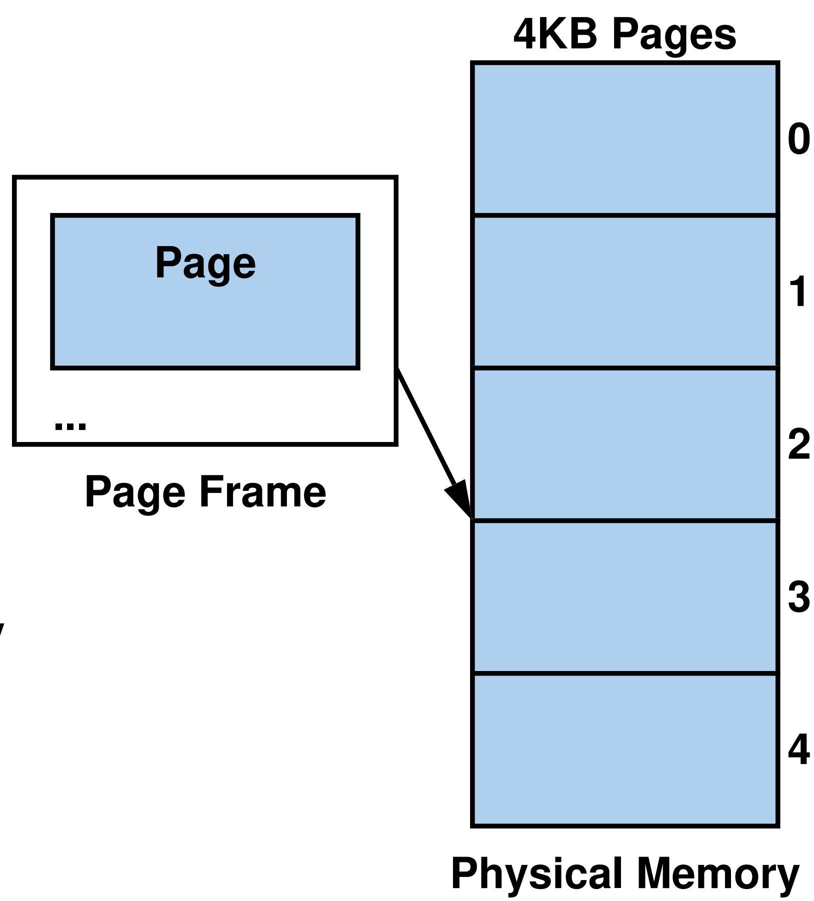
page 0	4KB Pages
page 1	4KB Pages
page 2	4KB Pages
...	...
page 262141	4KB Pages
page 262142	4KB Pages
page 262143	4KB Pages

1GB Physical Memory

Copyright © William C. Cheng

Managing Allocated Physical Pages: 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 *virtual address* for reading/writeing the page
  - contains a *physical page number*
  - there is a *one-to-one mapping* between page frames and allocated physical pages
    - we use "page frame" and "physical page" interchangeably
  - page frames can be *shared*
    - "physical pages" can be shared

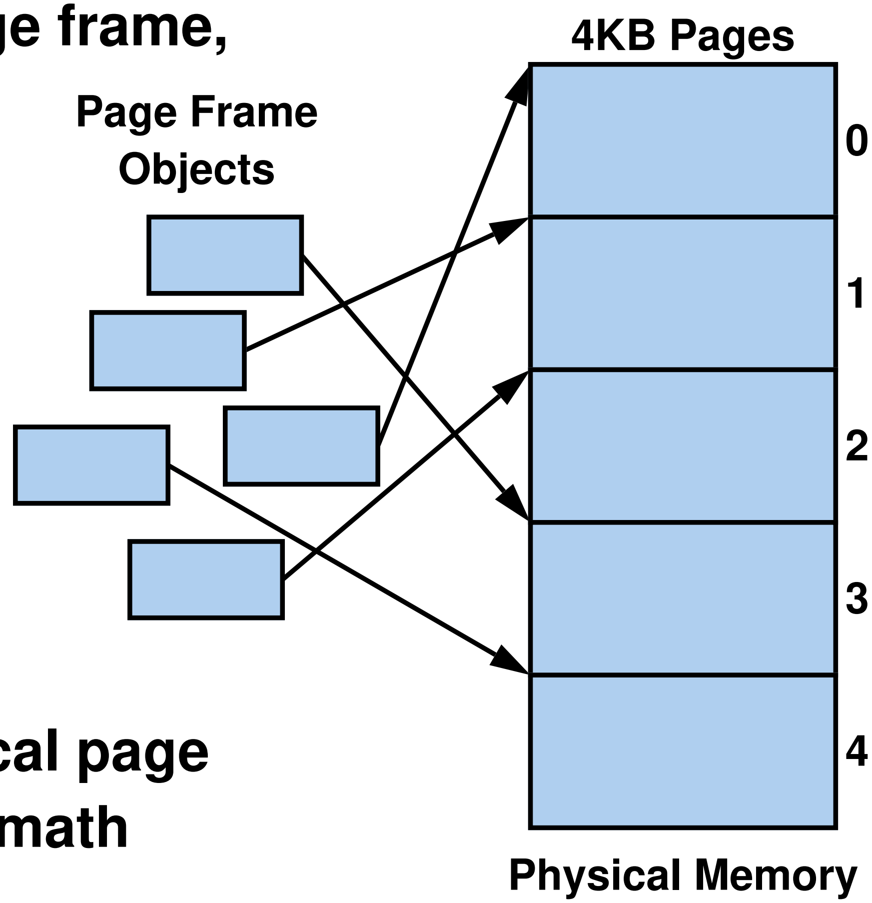


- Weenix note for kernel 3:
  - `pagenum` is not the "page number" mentioned here!

Copyright © William C. Cheng

Page Frames

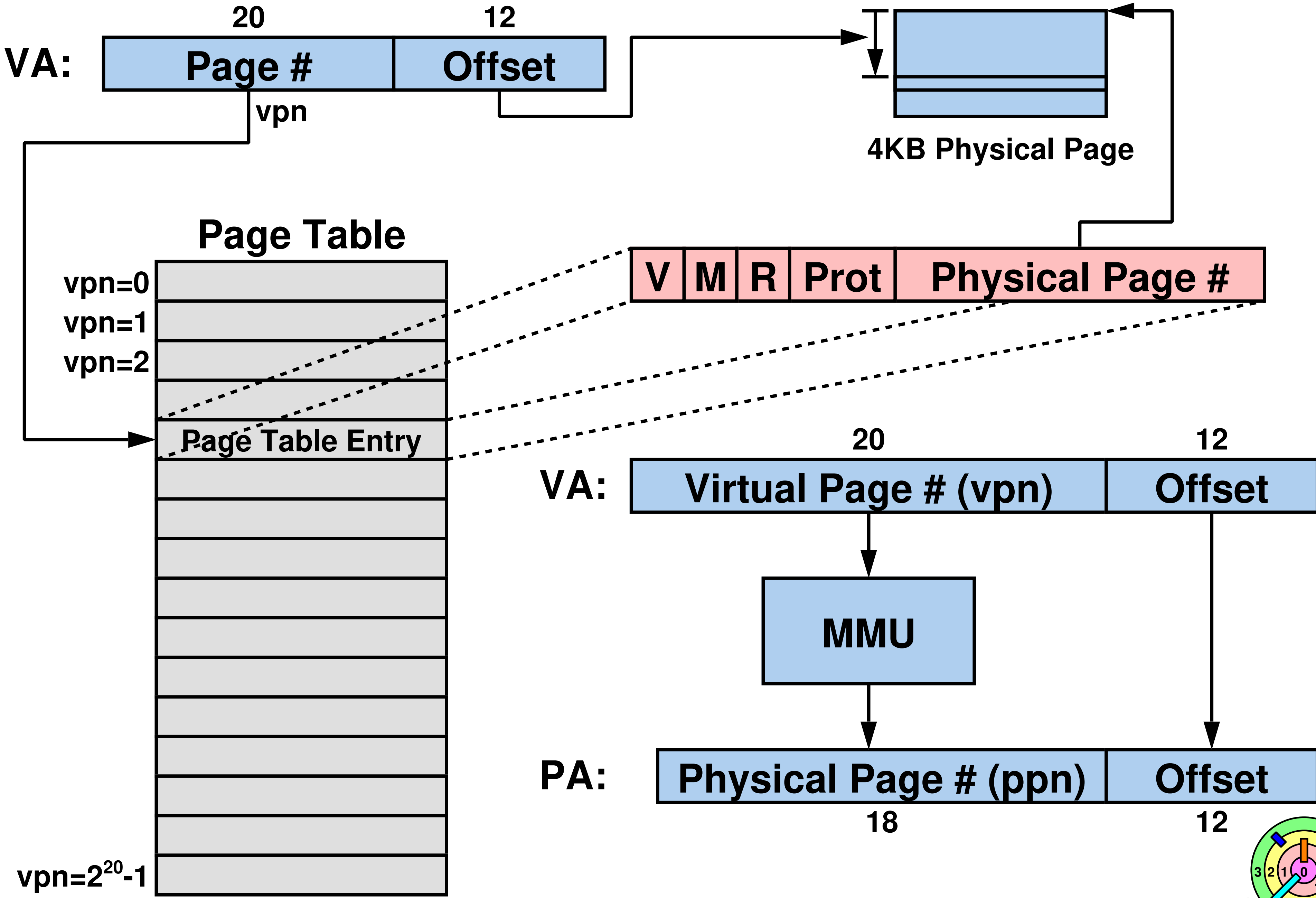
- It is important to be able to perform both *forward lookup* and *reverse lookup*
  - *forward* lookup: given a virtual address of a process, find page frame
  - *reverse* lookup: given a page frame, find processes and virtual addresses that map to this page frame



- `weenix` page frame data structure is a bit involved
  - the kernel must use a *virtual address* to write into a physical page or read the content of a physical page
    - usually requires pointer math
  - see kernel 3 FAQ

Copyright © William C. Cheng

Basic (Two-level) Page Tables



Copyright © William C. Cheng