Paging

Mendel Rosenblum

Virtual Memory

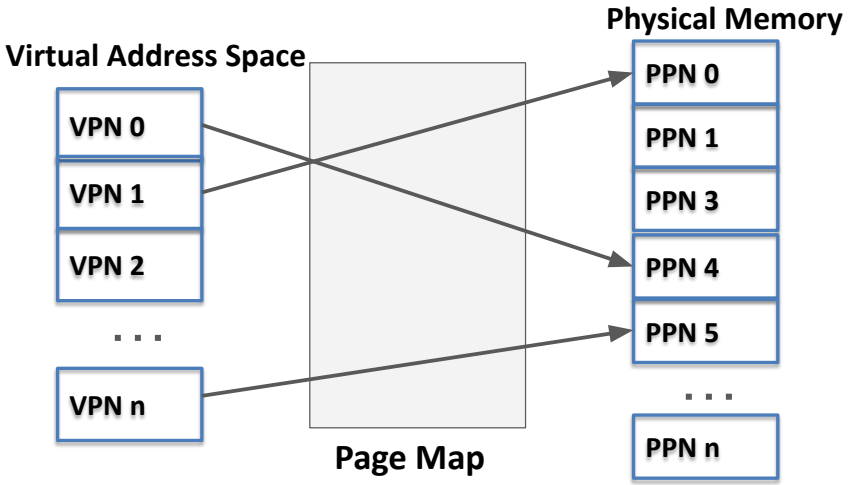
Optional readings: Operating Systems: Principles and Practice: Chapter 8

Key idea behind paging: Pages

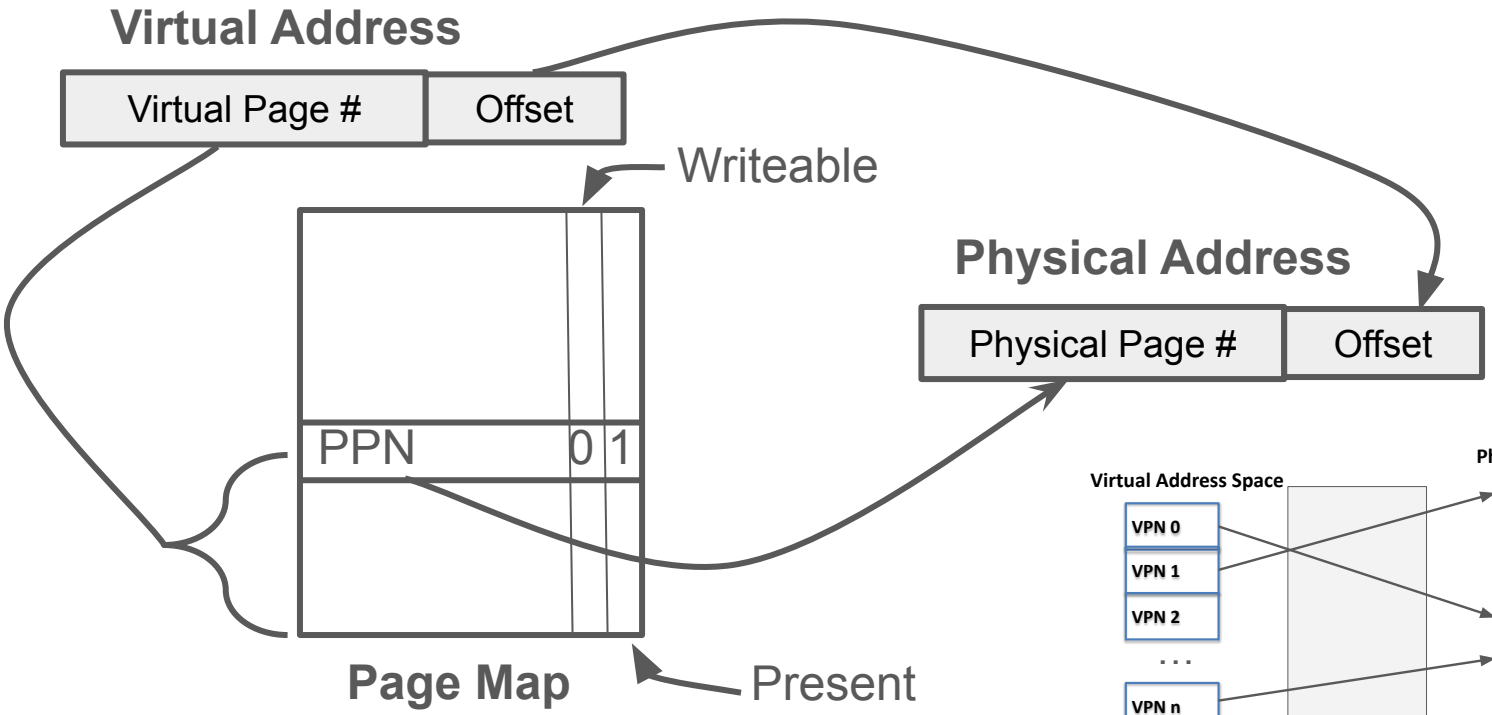
- Divide address space into fixed sized chunks (**pages**)
  - Done to both the virtual and physical address space
  - 4KB (x86 myth) and 16KB (Macbook) sizes commonly used today

Paging

- MMU needs **Page Map** (also called called a **Page Table**)
  - Maps virtual page to physical page



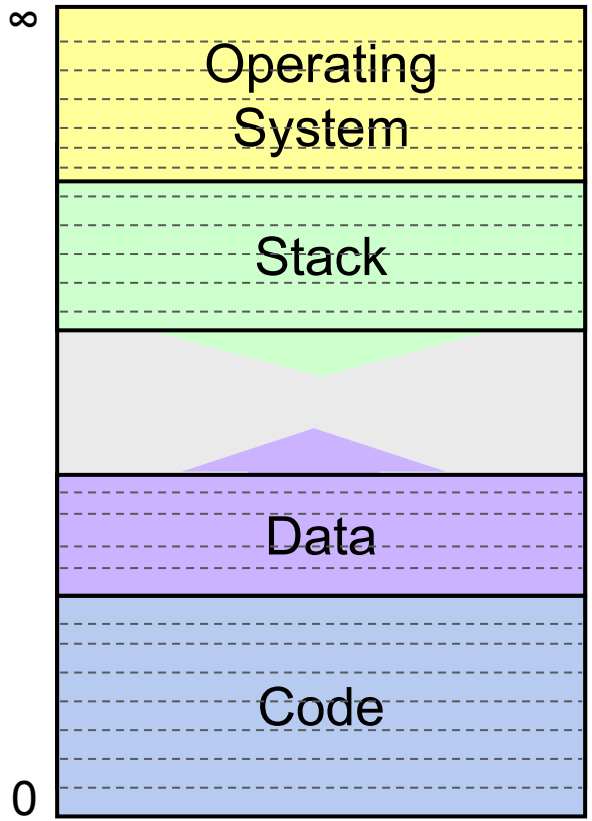
Page map / Page Table



Fixed size makes memory management easier

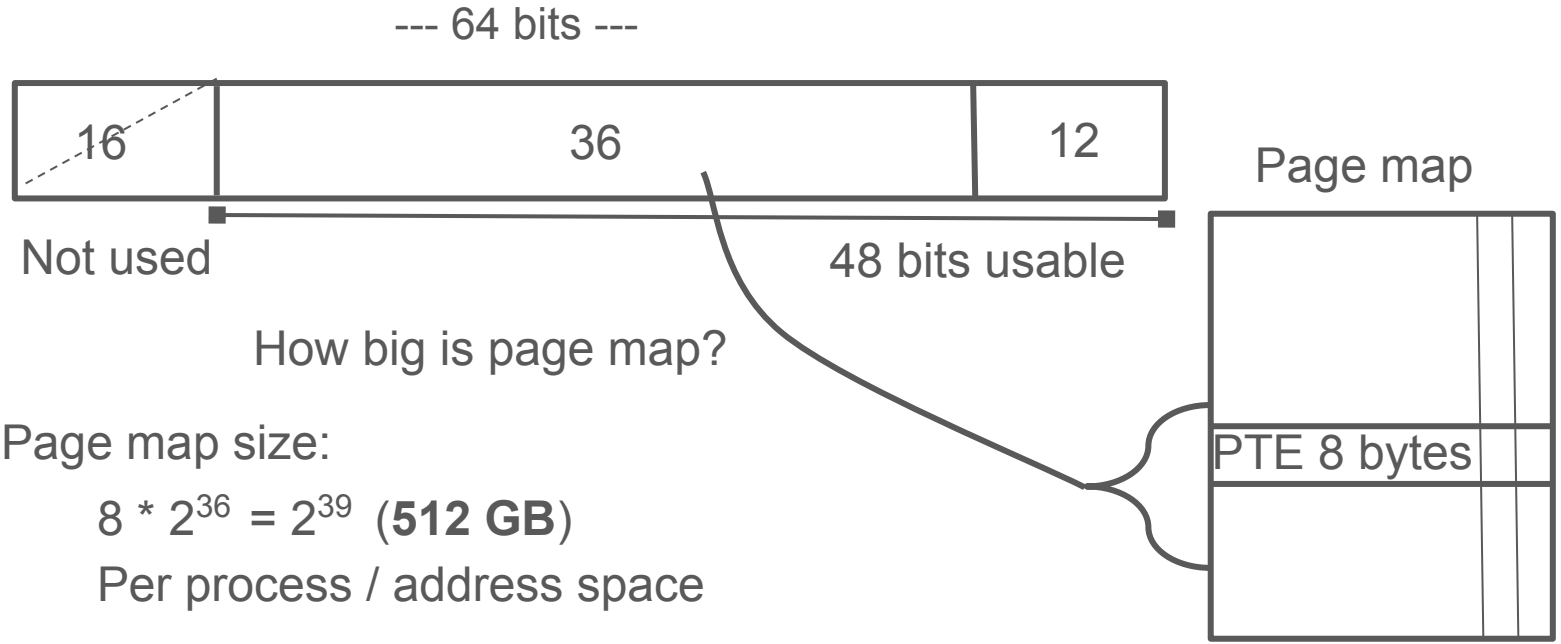
- Physical memory management
  - OS maintains free list of available physical memory pages
    - Allocation: remove from list
    - Freeing: add to list
- Virtual address space management
  - Segment: collection of pages of size desired by program
    - Start at any page boundray

Virtual Address space pages



Big challenge: Page map implementation

Example: x86-64 (myth machines) 64 bit virtual address, 4KB page size



How big is page map?

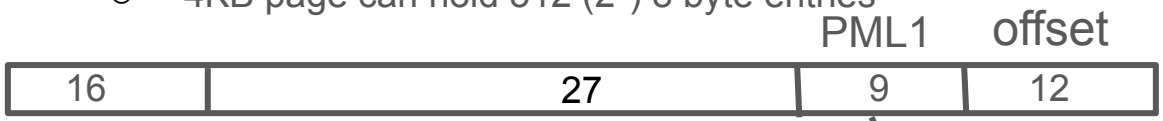
Page map size:

$8 * 2^{36} = 2^{39}$ (**512 GB**)

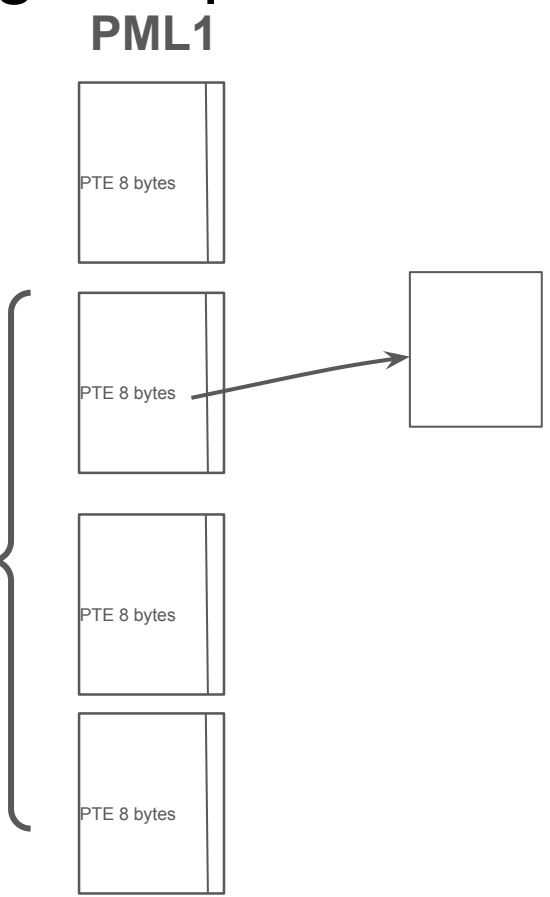
Per process / address space

Approach used in x86-64 page map

- Break page map into page sized (4KB) chunks
  - 4KB page can hold 512 (2^9) 8 byte entries

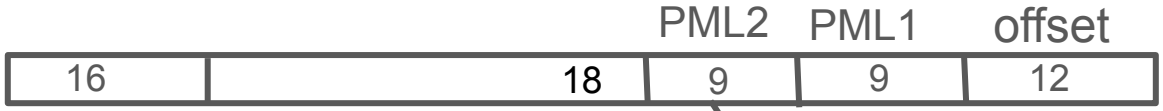


- 2^{27} is still too large: Add another PML
- **P**age **M**ap **L**evel - PML

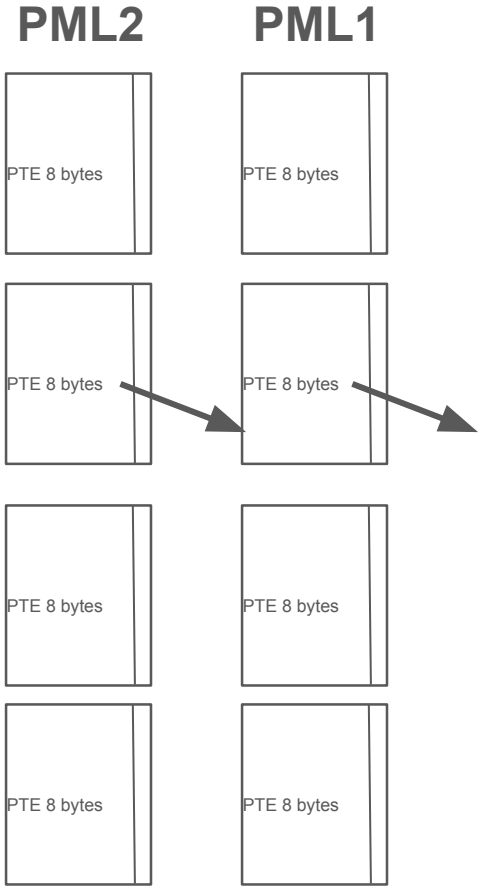


More Page Map Levels

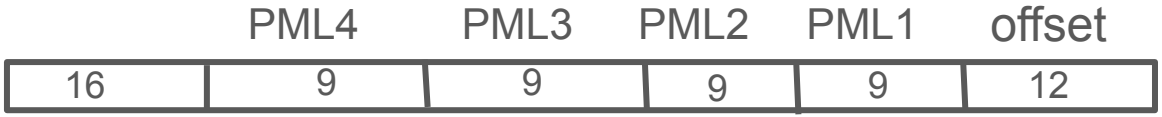
- Add a Page Map Level 2 (PML2)



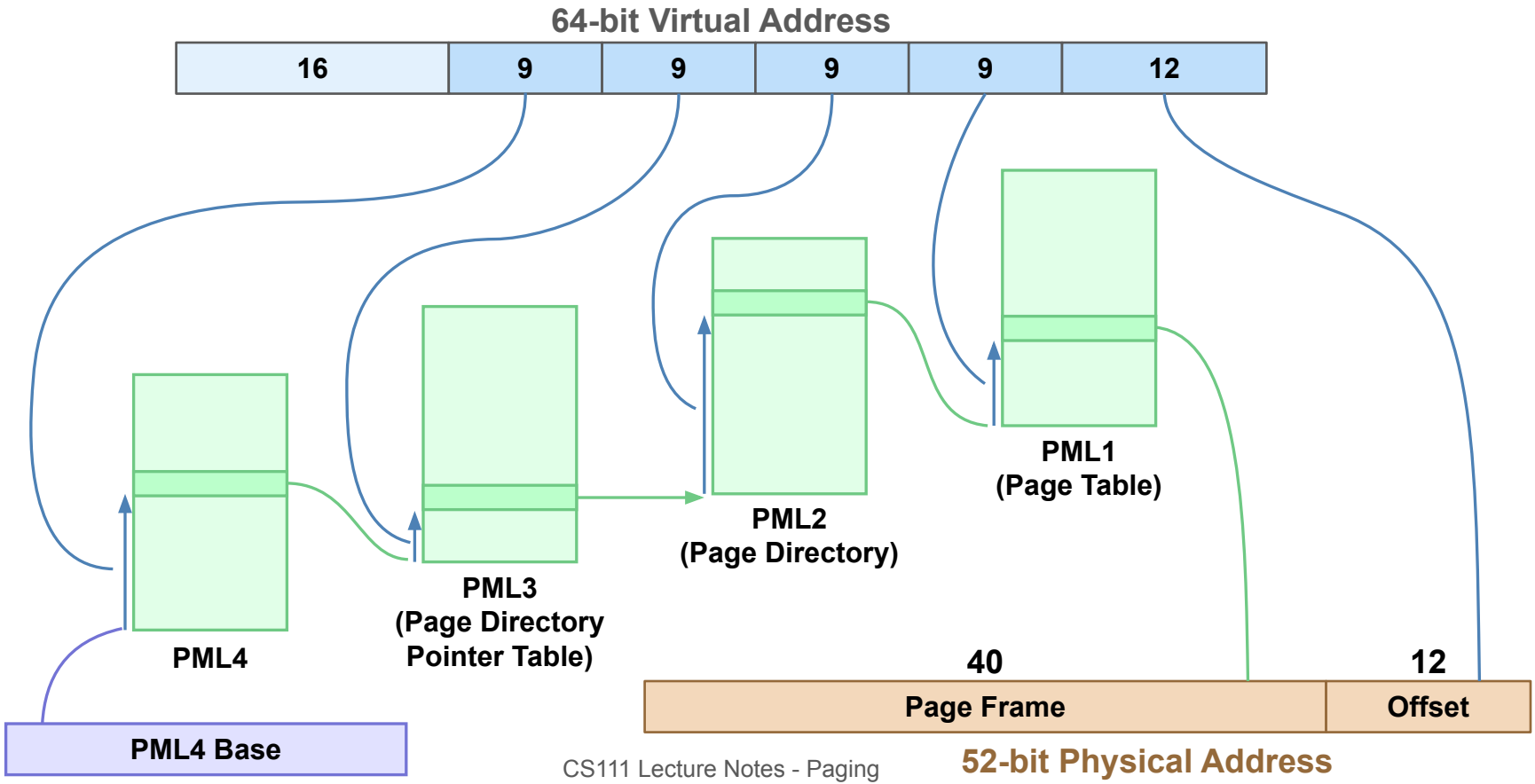
Has present bits so we don't need all PML1 pages



- Keep adding pmap levels:

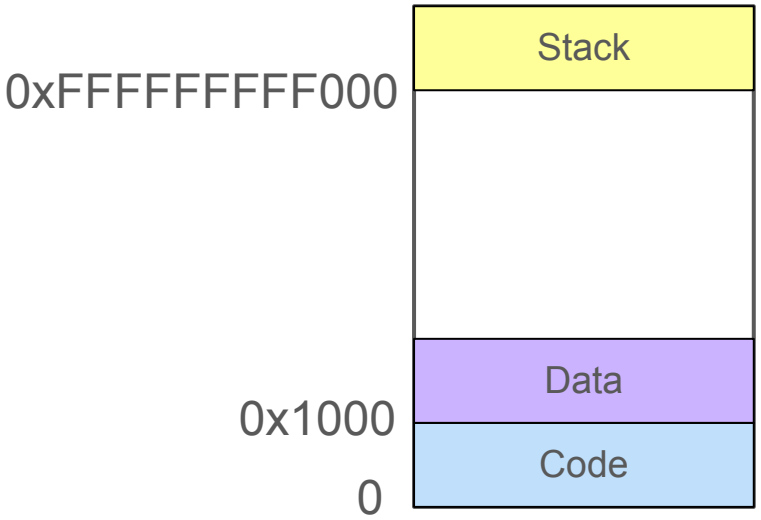


x86-64 Address Translation



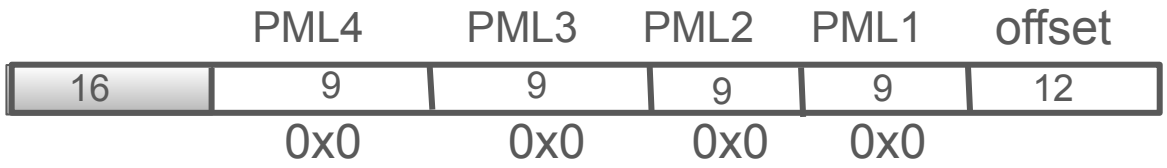
Virtual address space example

- One page of code at 0x0000
- One page of data at 0x1000
- One page of stack at top (0xFFFFFFFFF000)

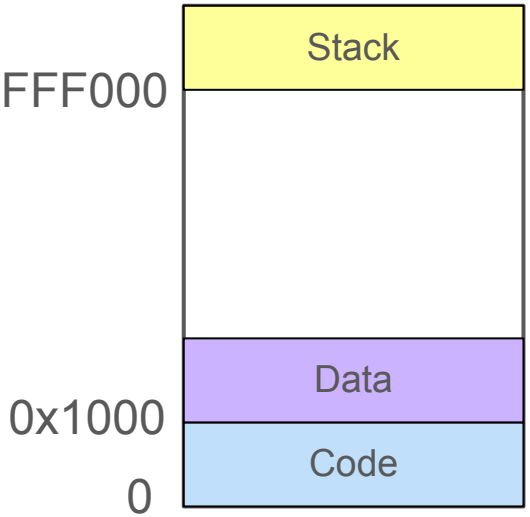


Access some instruction in code page

Access in the code page 0x0 - 0xFFF

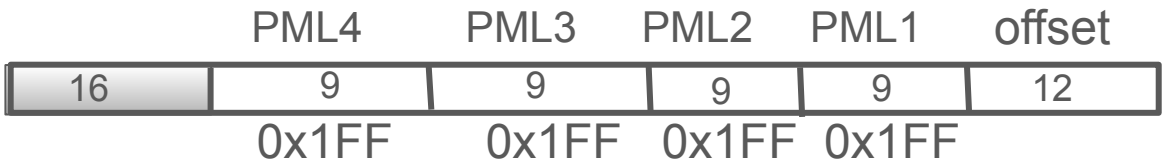


1. Goto PML4 index 0x0 extract PML3 start addr
   PML4 Base register points to PML4 start
2. Goto PML3 start addr index 0x0 extract PML2 start addr
3. Goto PML2 start addr index 0x0 extract PML1 start addr
4. Goto PML1 start addr index 0x0, extract PTE (PPN)

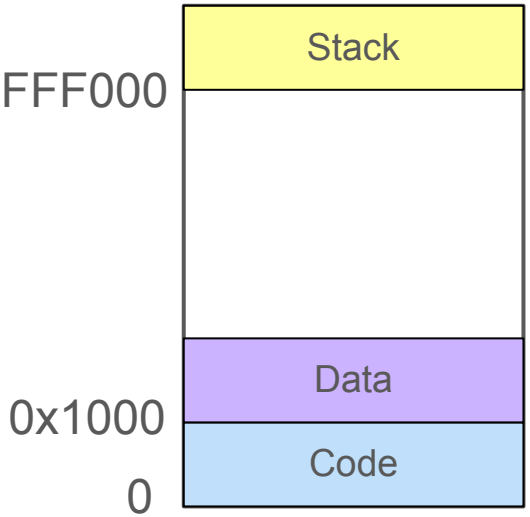


Access to the stack page

Access in the stack page 0xFFFFFFFFF000 - 0xFFFFFFFFFFFF (e.g. 0x4)

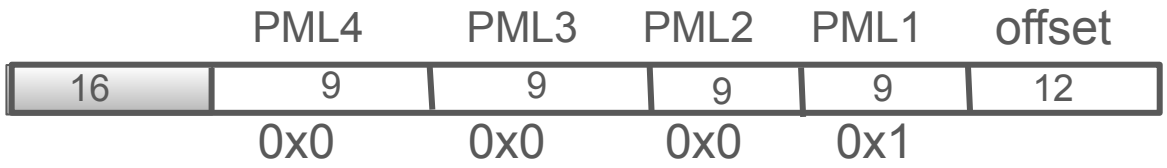


1. Goto PML4 index 0x1FF (512) extract PML3 start addr
   PML4 Base register points to PML4 start addr
2. Goto PML3 start addr index 0x1FF extract PML2 start addr
3. Goto PML2 start addr index 0x1FF extract PML1 start addr
4. Goto PML1 start addr index 0x1FF, extract PTE (PPN)

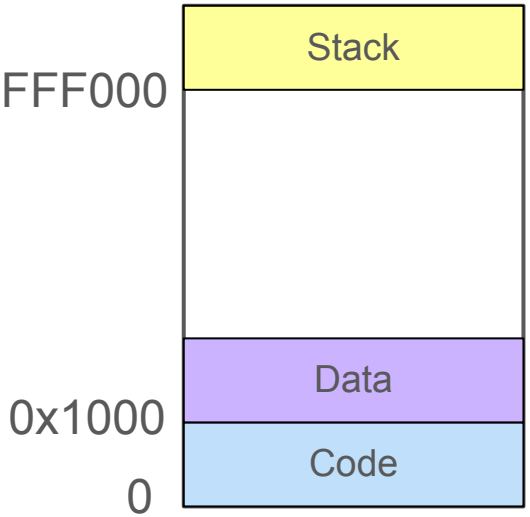


Access to the data page

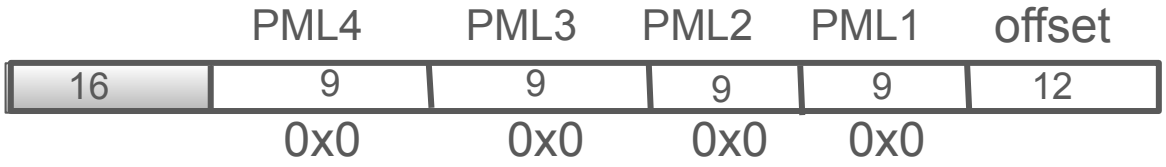
Access in the data page 0x1000 - 0x1FFF (e.g. 0x1008)

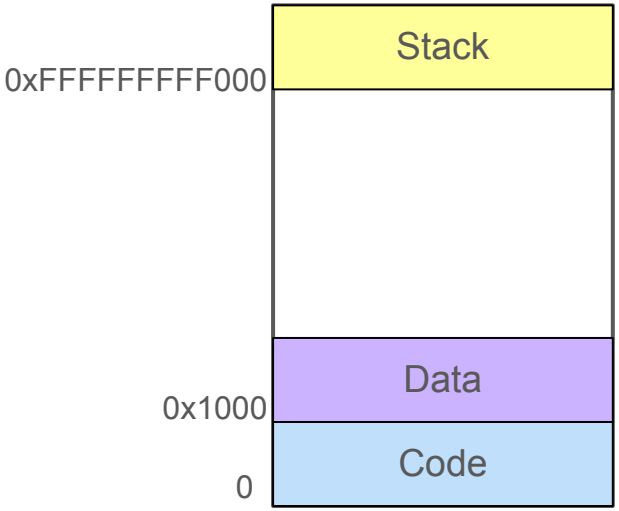


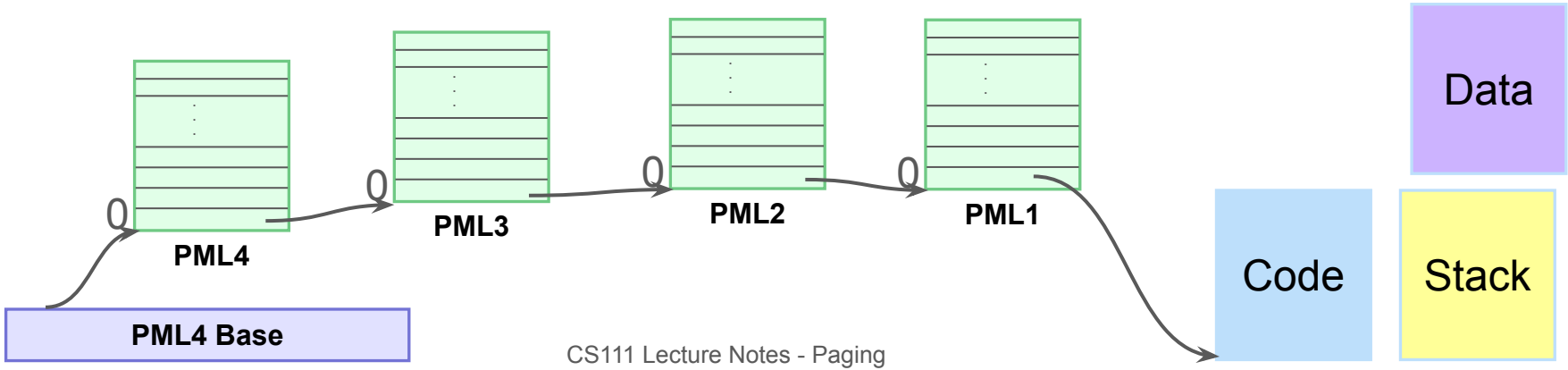
1. Goto PML4 index 0x0 extract PML3 start addr
   PML4 Base register points to PML4 start addr
2. Goto PML3 start addr index 0x0 extract PML2 start addr
3. Goto PML2 start addr index 0x0 extract PML1 start addr
4. Goto PML1 start addr index 0x1, extract PTE (PPN)



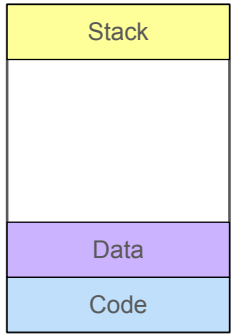
Access address 0x0 (Code)

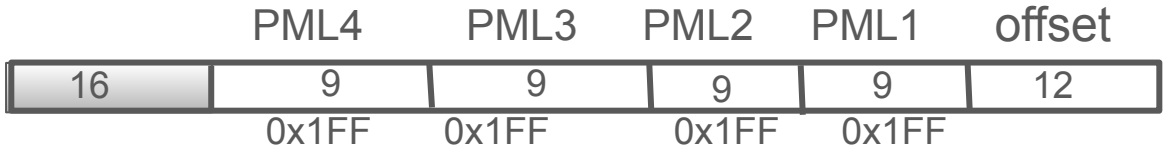


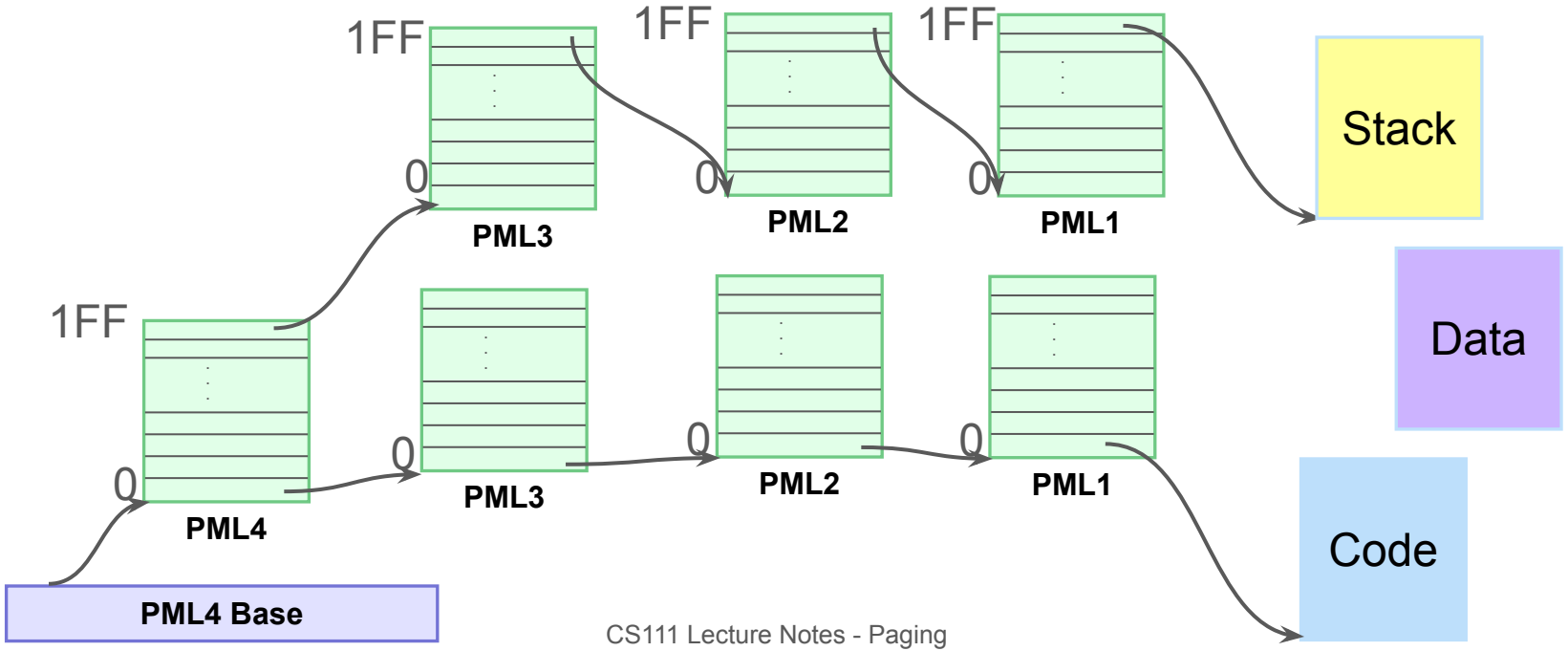




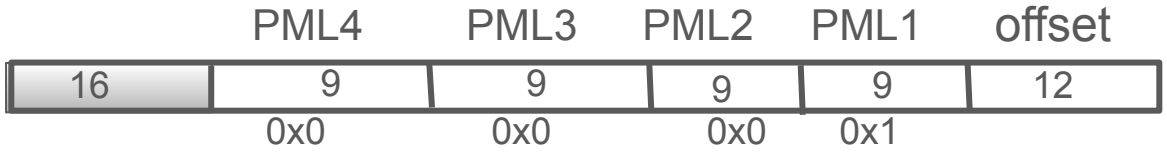
Access address 0xFFFFFFFFF000 (Stack)

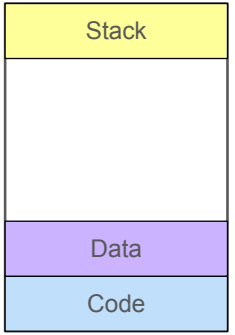


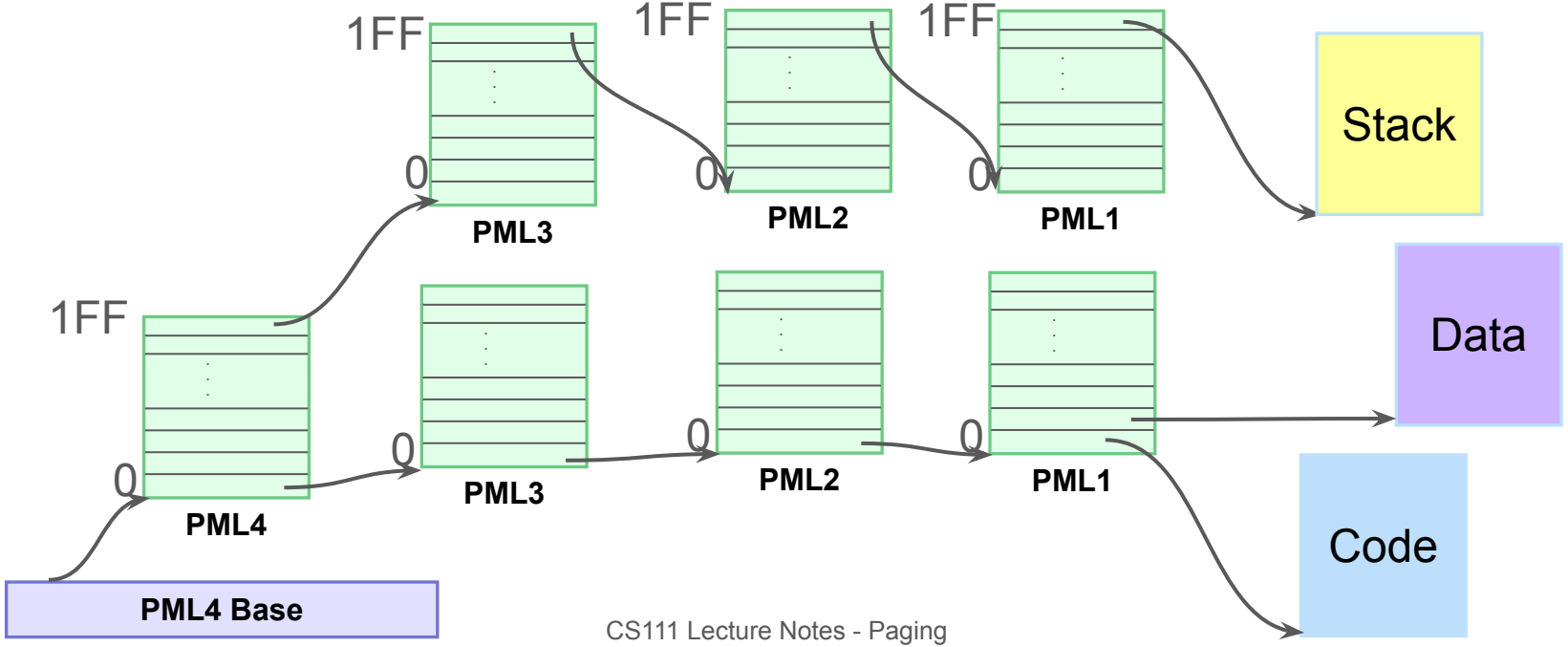




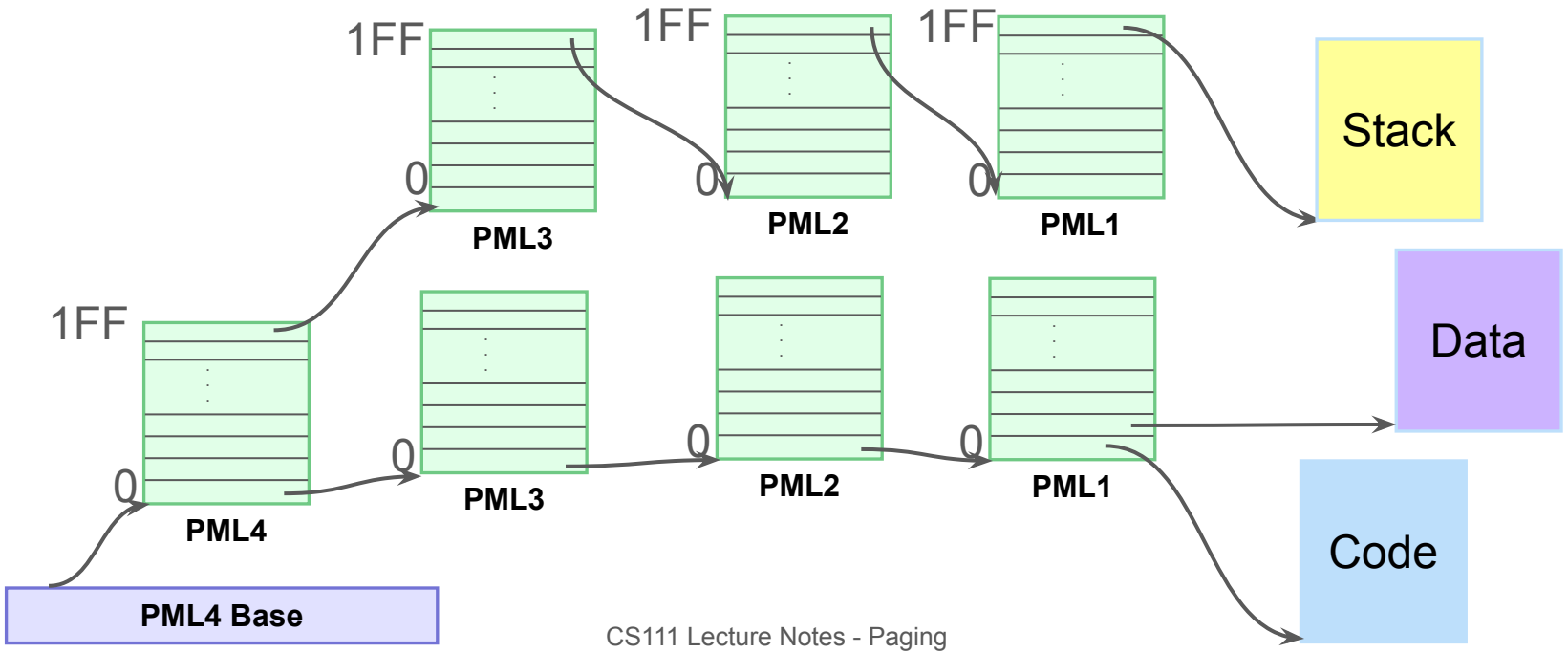
Access address 0x1000 (Data)







Remind you of any data structure you learned?



Sharing memory between processes

With X86-64:

- A single 4KB page (two PML1 pointing at same physical page)
- An entire page table at any level
  - In a PML2 - PML1 sized - 512*4K 2MB shared object
  - In a PML3 - PML2 sized - 512*512*4K 1GB shared object
  - In a PML4 - PML3 sized - 512*512*512*4K 512GB shared object
- Possible to support different sized pages:
  - 4K, 2MB, 1GB

Big issue: Too many memory accesses

- Page maps are too large to load into fast memory in the MMU
  - Page maps kept in main memory
  - MMU holds base address for top-level page map
- Taking 4 memory references to walk PMLx pages unacceptable
- Solution: Cache VPN ⇒ PPN mappings in MMU
  - Given the name **Translation Lookaside Buffer** or **TLB**

Translation Lookaside Buffer (TLB)

- Small cache in MMU of recent translations
  - Typical TLB sizes: 64-2048 entries
  - Cache entry contains:
    - Page number of a virtual address (VPN) (36 bits for x86-64)
    - Physical page number (PPN) (40 bits for x86-64)
- Needs to be super fast and high hit rate
  - Fully associative design
  - TLB "hit rates" typically 95% or more. Much locality in memory references
- Operation: CPU issues a memory op specifying virtual address (VPN | offset)
  - Hit in TLB (VPN matches) use PPN and check protection bits
  - Miss in TLB: walk page map and fill TLB with VPN, PPN, and protection bits

TLB OS complications

- TLB is mostly transparent to the OS

Except:

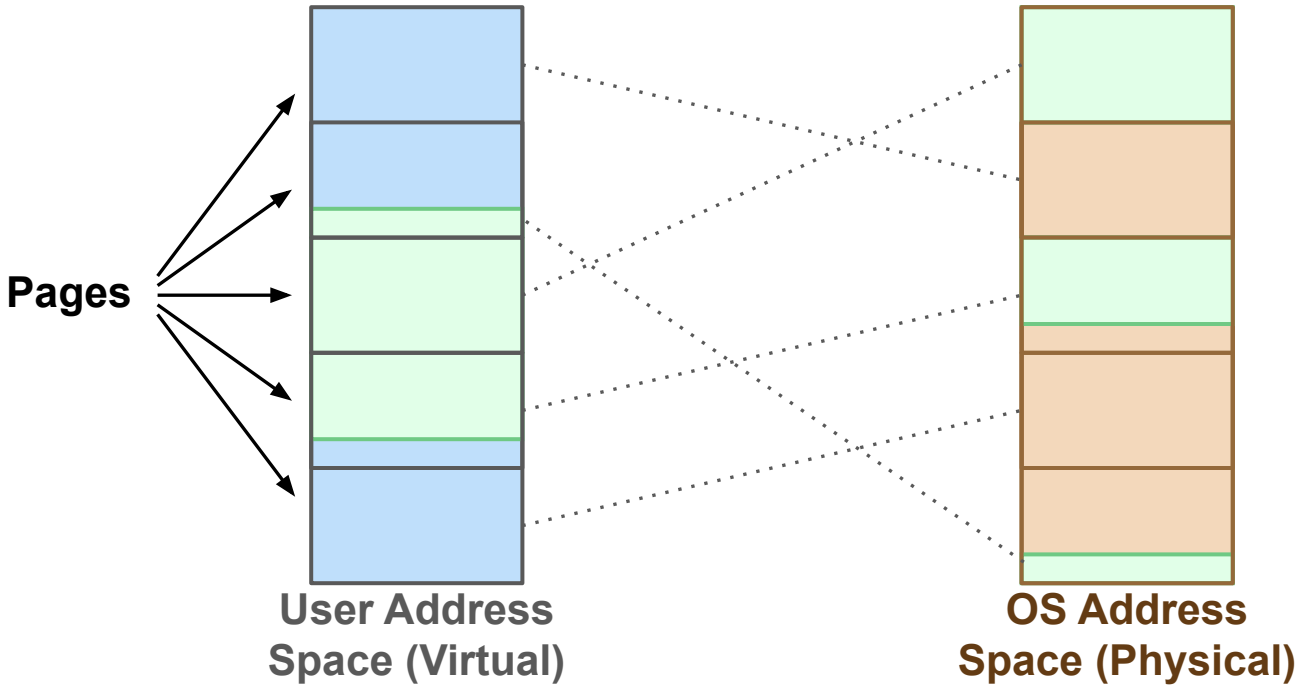
- Switching page tables (context switch) must invalidate cache
  - X86-64 assignment to PML4 Base flushes TLB
  - Some architectures will put a PID in TLB entry to avoid flush
- Changing a page map might require invaliding a TLB entry
  - X86-64: INVLPG instruction

OS access to user memory

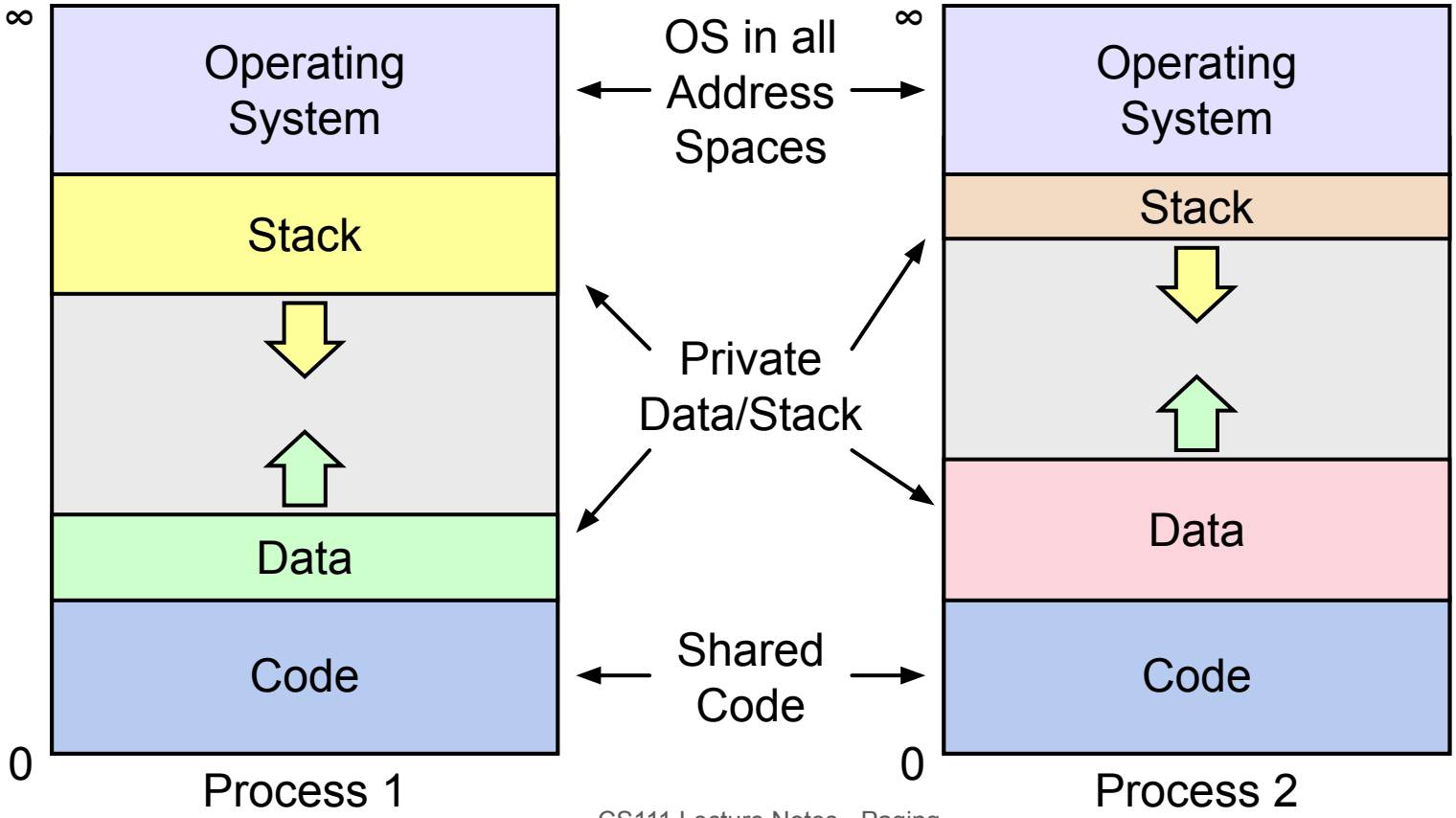
- Two approach:
  - Run OS unmapped
    - Data structures contiguous in virtual memory likely not contiguous in physical memory
    - OS needs to translates user addresses in software
  - Run OS mapped into virtual address space
    - Still issue with I/O devices

Accessing User Memory: Umapped OS

write(fd, buffer, length)

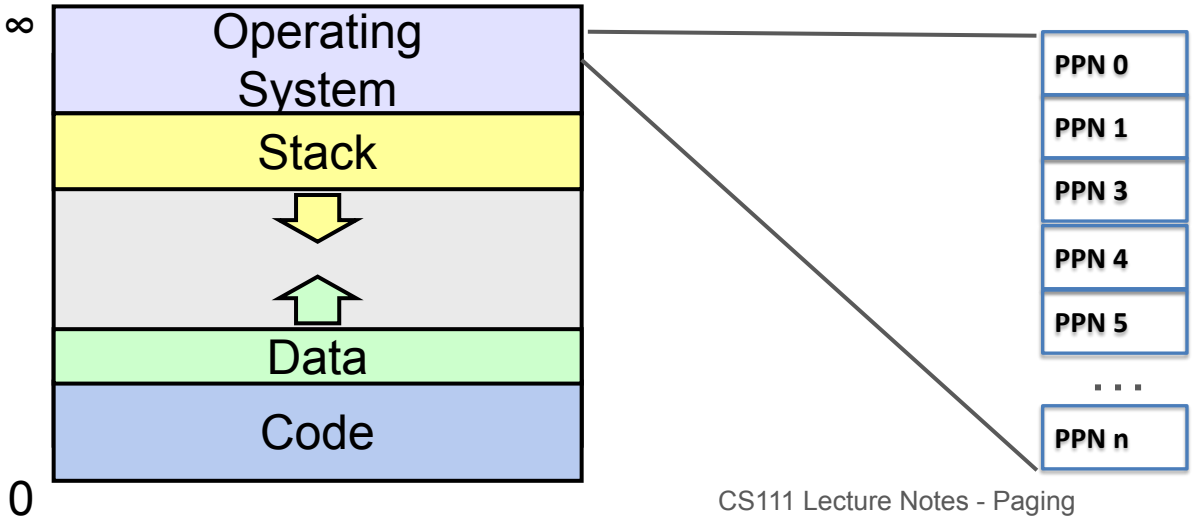


OS and User in Same Address Space



Memory aliasing

- Memory aliasing: Multiple virtual addresses for same physical page
  - Easy to do: duplicate a PML1 PTE
- Weird for a user process (two copies of the same variable)
- Commonly done in an OS to make physical memory available easily



Fragmentation: internal and external

- Still can have memory inside a page not being used
  - Call **internal fragmentation**
- Virtual memory get rid of fragment between processes
  - Call **external fragmentation**
- Typically the last page of segment has some internal fragmentation
  - Since segments are large, internal fragmentation is small
  - Larger page sizes can increase internal fragmentation