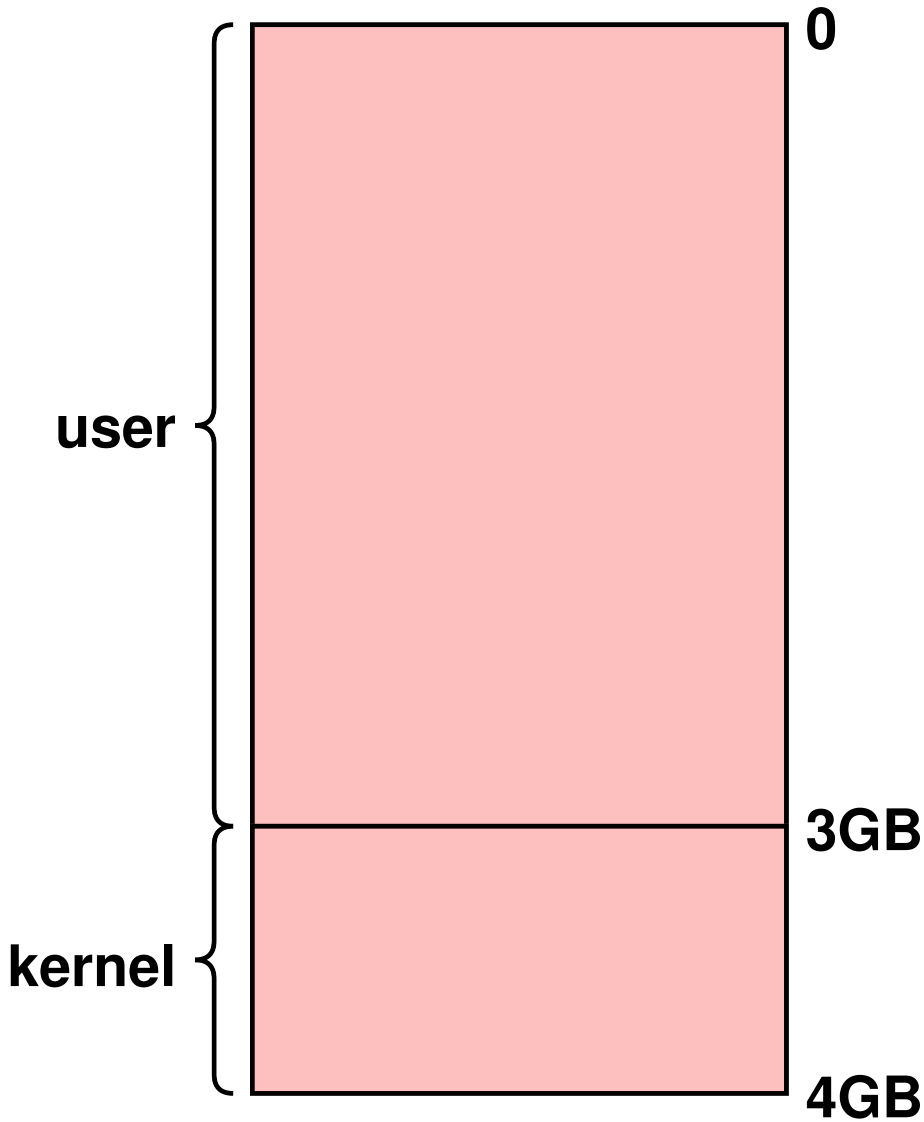
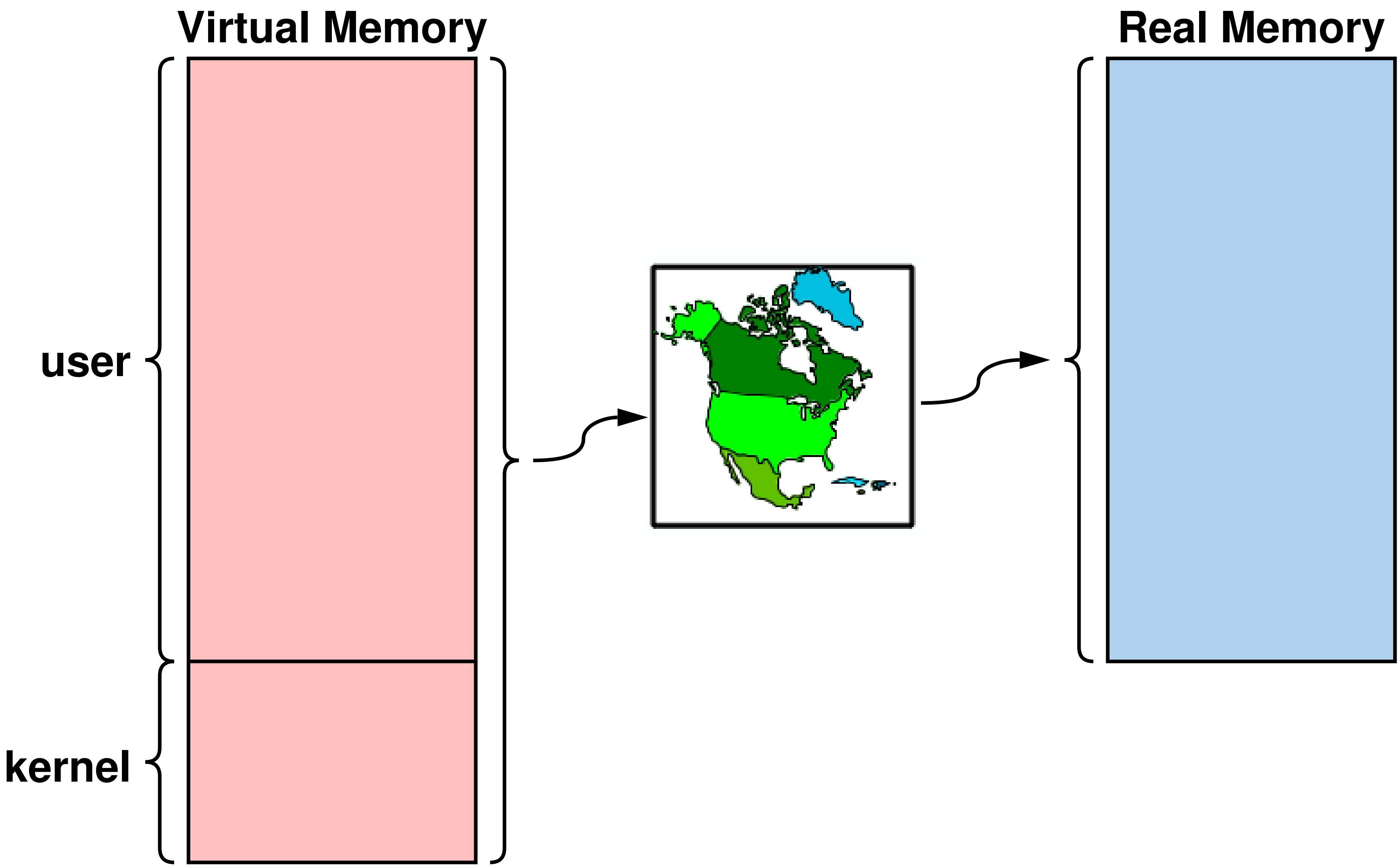
7.3 Operating System Issues

- General Concerns
- *Representative Systems*
- Copy on Write and Fork
- Backing Store Issues

Linux Intel x86 VM Layout



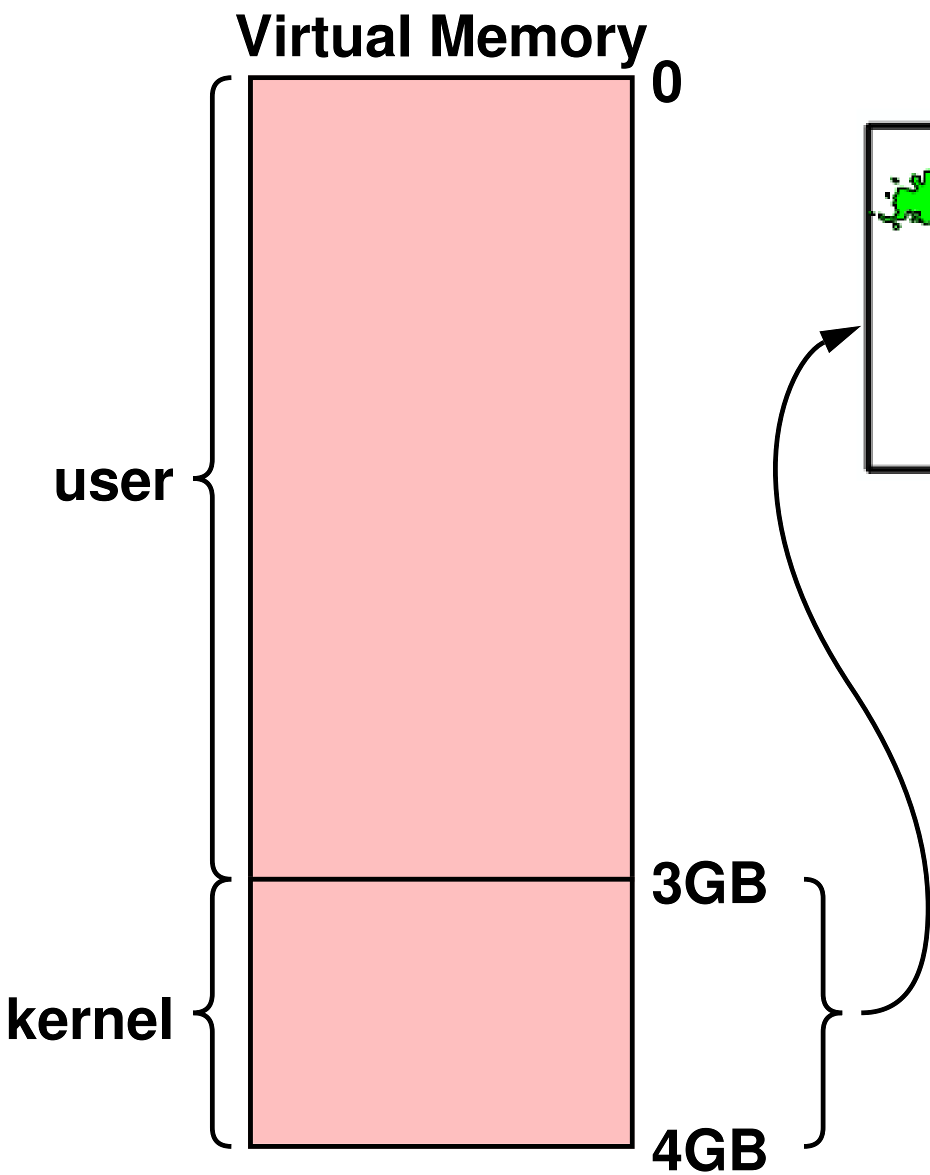
Real Memory

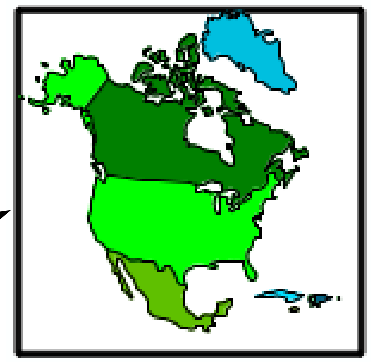


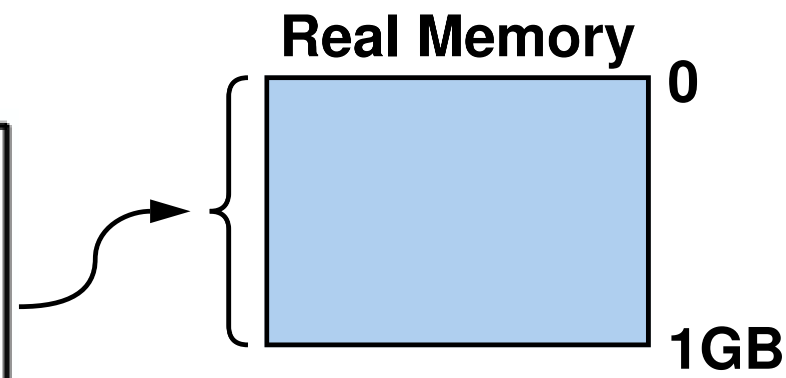
Memory Allocation

- User
  - virtual allocation
    - fork
    - pthread_create
    - exec
    - brk
    - mmap
  - real allocation
    - (not done)
- OS kernel
  - virtual allocation
    - fork, etc.
    - some kernel data structures
    - pretty much any time when you allocate from a slab allocator
  - real allocation
    - page faults
    - some kernel data structures
      - e.g., page tables
    - pretty much any time when you allocate from the buddy system

Example: 1GB Of Real Memory

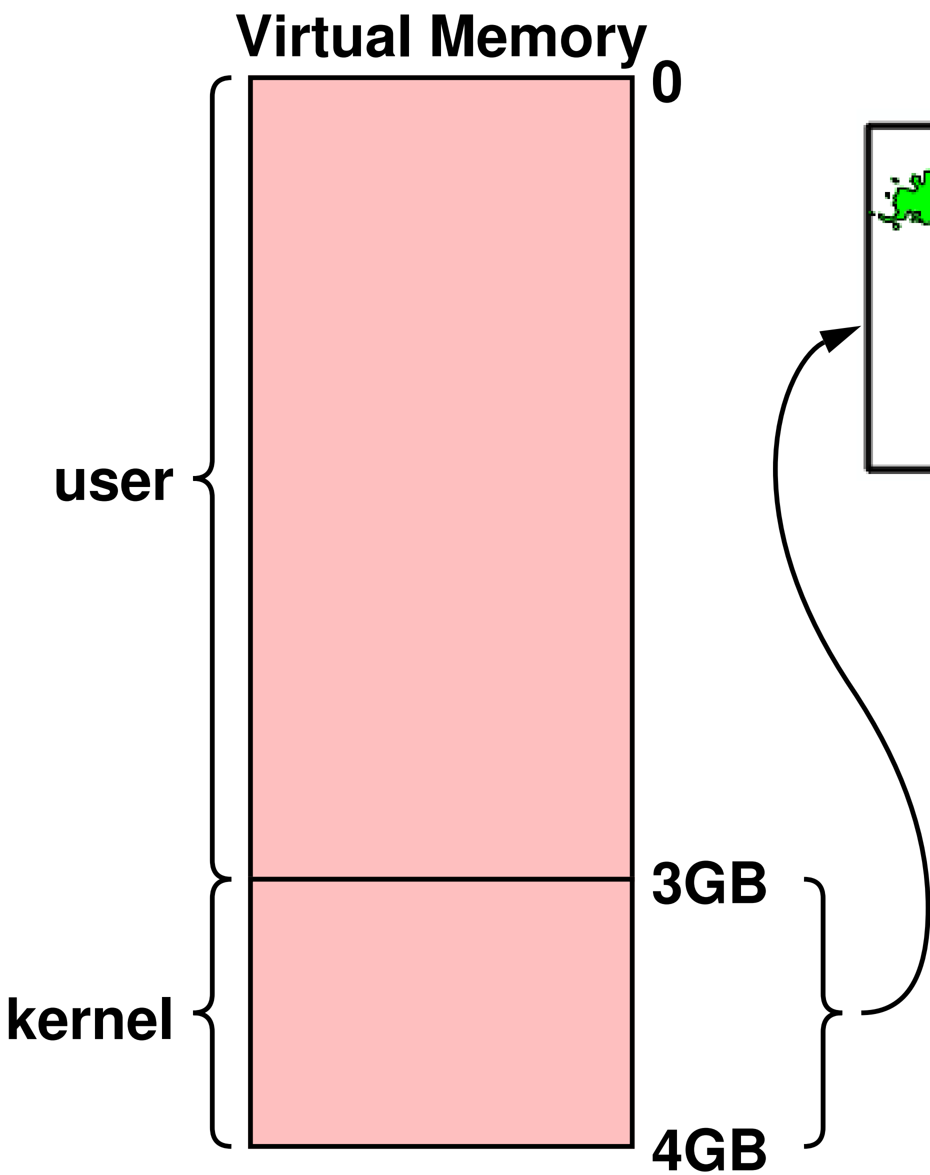


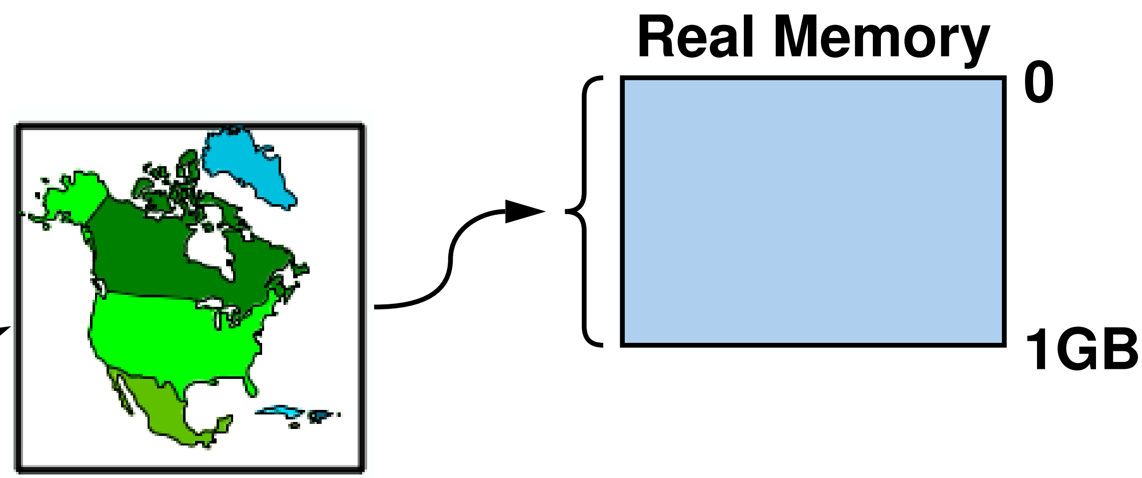




- If you only have 1GB of physical memory
  - conceptually, can setup page table so that physical address = kernel vaddr - `0xc0000000`
    - *not done this way in real systems*
  - OS can read any *physical memory location* easily

Example: 1GB Of Real Memory

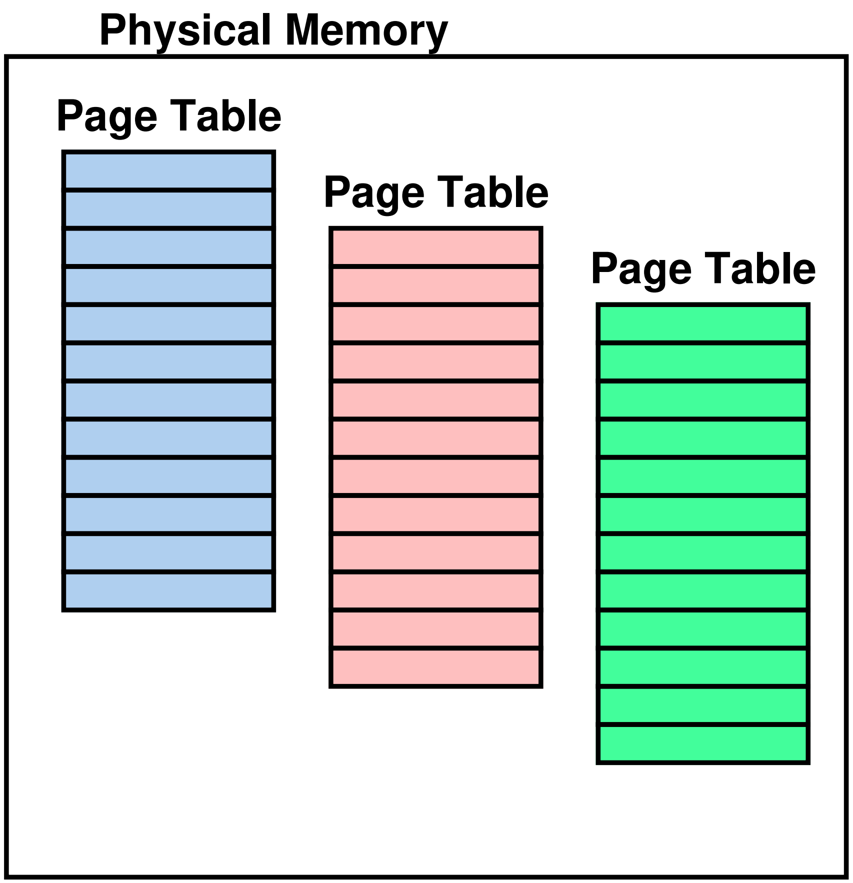




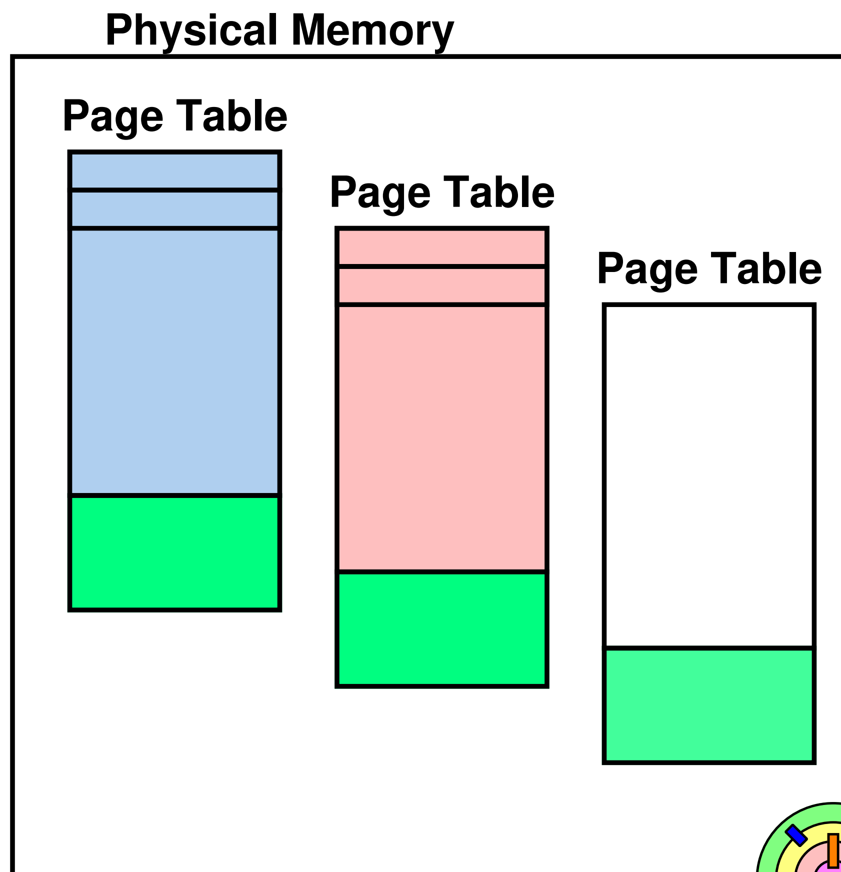
- When allocating page frames for user space memory pages
  - these pages are mapped from *both* user and kernel spaces
  - OS can read any ***user space memory location*** directly (assuming it's "mapped")

Example: 1GB Of Real Memory

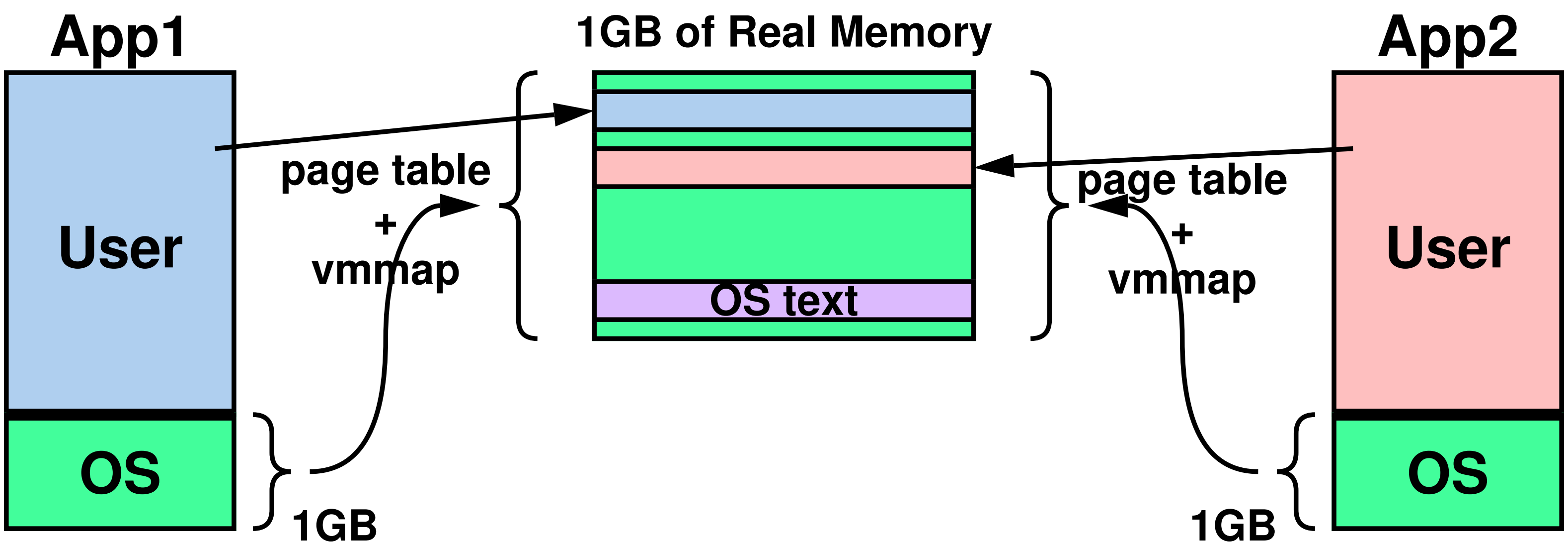
- **What a user thread becomes the kernel thread, it's still in the *same process*, therefore, should use the *same page table***
- **Multiple processes - page tables**
  - **does *not* look like this**



  - **but look like this:**



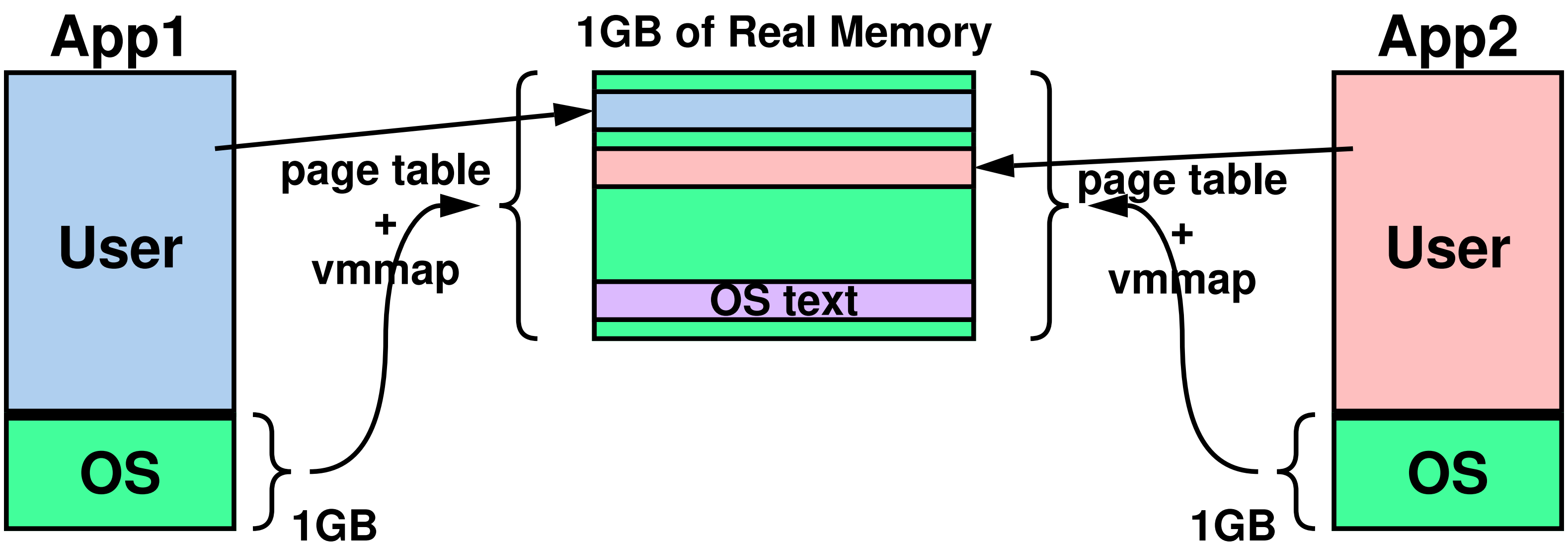
Example: 1GB Of Real Memory



- When you switch from one process to another
  - OS code and OS data stay exactly where they were
  - bottom 1/4 of page tables of all processes are *mapped identically*
    - for kernel-only processes, only the bottom 1/4 of the page tables are mapped (i.e., top 3/4 always have V=0)

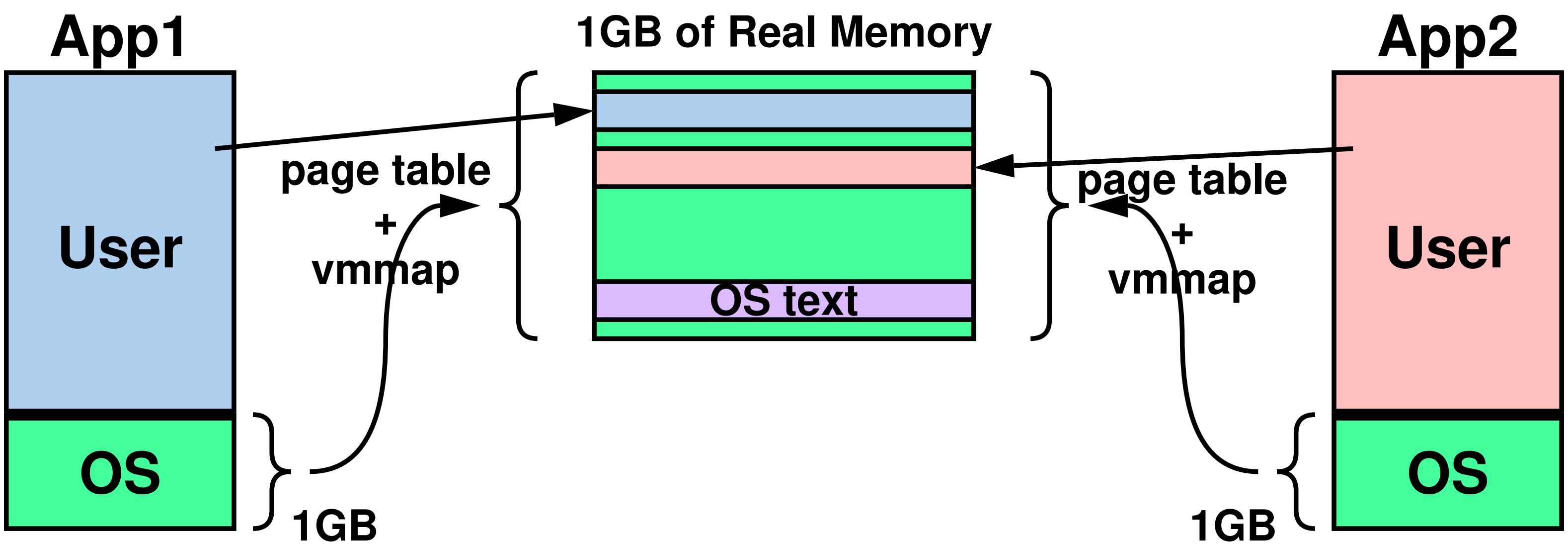
- How to setup top 3/4 page table for a user process?
  - by using the *vmmap* (virtual memory map) data structure
    - *vmmap* is only needed to manage *user portion of the address space*

Example: 1GB Of Real Memory



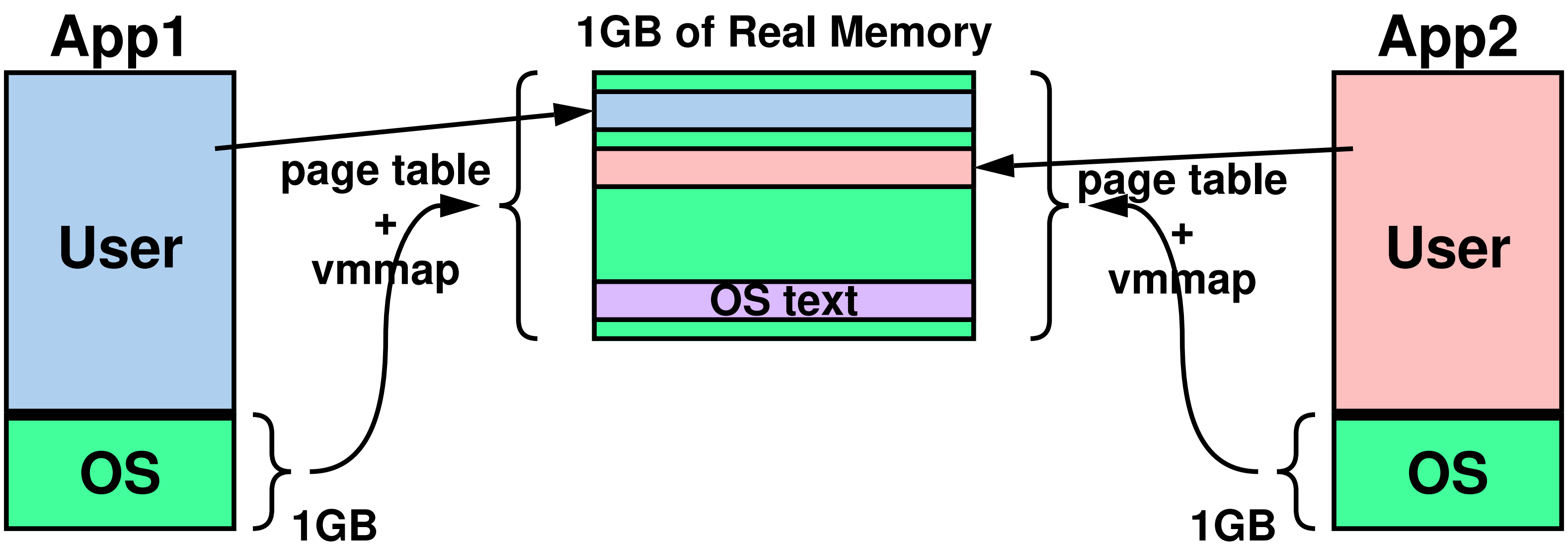
- ***Every physical address*** that's allocated to a user process can have ***two virtual addresses***
  - one for the kernel and one for a user process
  - which virtual address should the kernel use?
    - be careful with user virtual address
      - if V bit in PTE is 0, cannot use such user virtual address in the kernel
    - can always use the kernel virtual address

Example: 1GB Of Real Memory



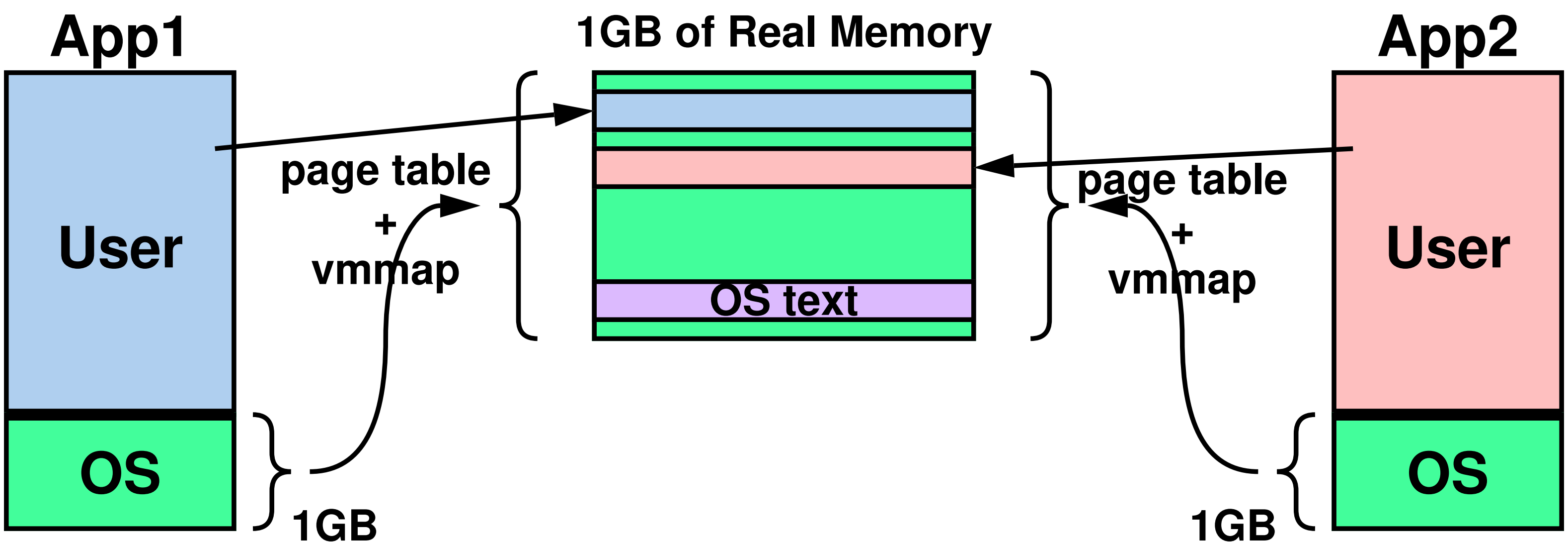
- When you trap into the kernel
  - you are still the *same thread* from *same process*
  - you use the *same page table*
    - in x86, there's a user/kernel bit in each PTE (page table entry)
      - top 3/4 of the PTEs set the bit to *U(ser)=1*
      - bottom 1/4 of the PTEs set the bit to *0 (Superviser)*
    - the *kernel part* of every page table are mapped *identically*
      - if you have 1GB or less physical memory, once this part is mapped, they will *never change*

Example: 1GB Of Real Memory



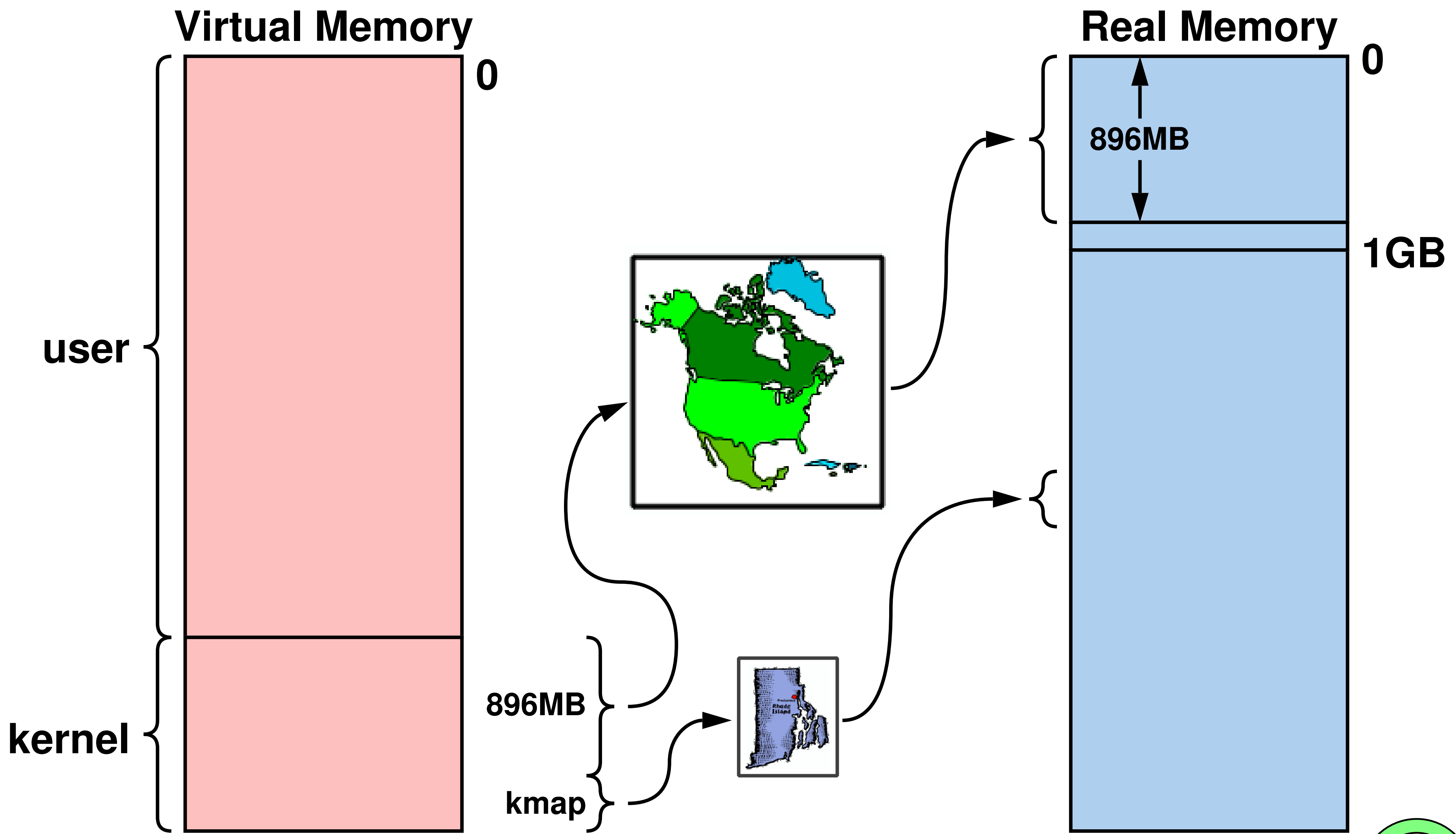
- HW: read `pt_init()` in "`kernel/mm/pagetable.c`" of `weenix`
  - kernel text, data, and bss starts at virtual address `0xc0000000`
  - then comes kernel's *page directory table* (4KB+4KB)
  - then comes kernel's *page tables* (4KB each)
  - understand how the first page table is setup for the *kernel*
  - understand that the *kernel*, just like user processes, can only use *virtual addresses!*
- Although `weenix` only has 256MB of physical memory

Example: 1GB Of Real Memory

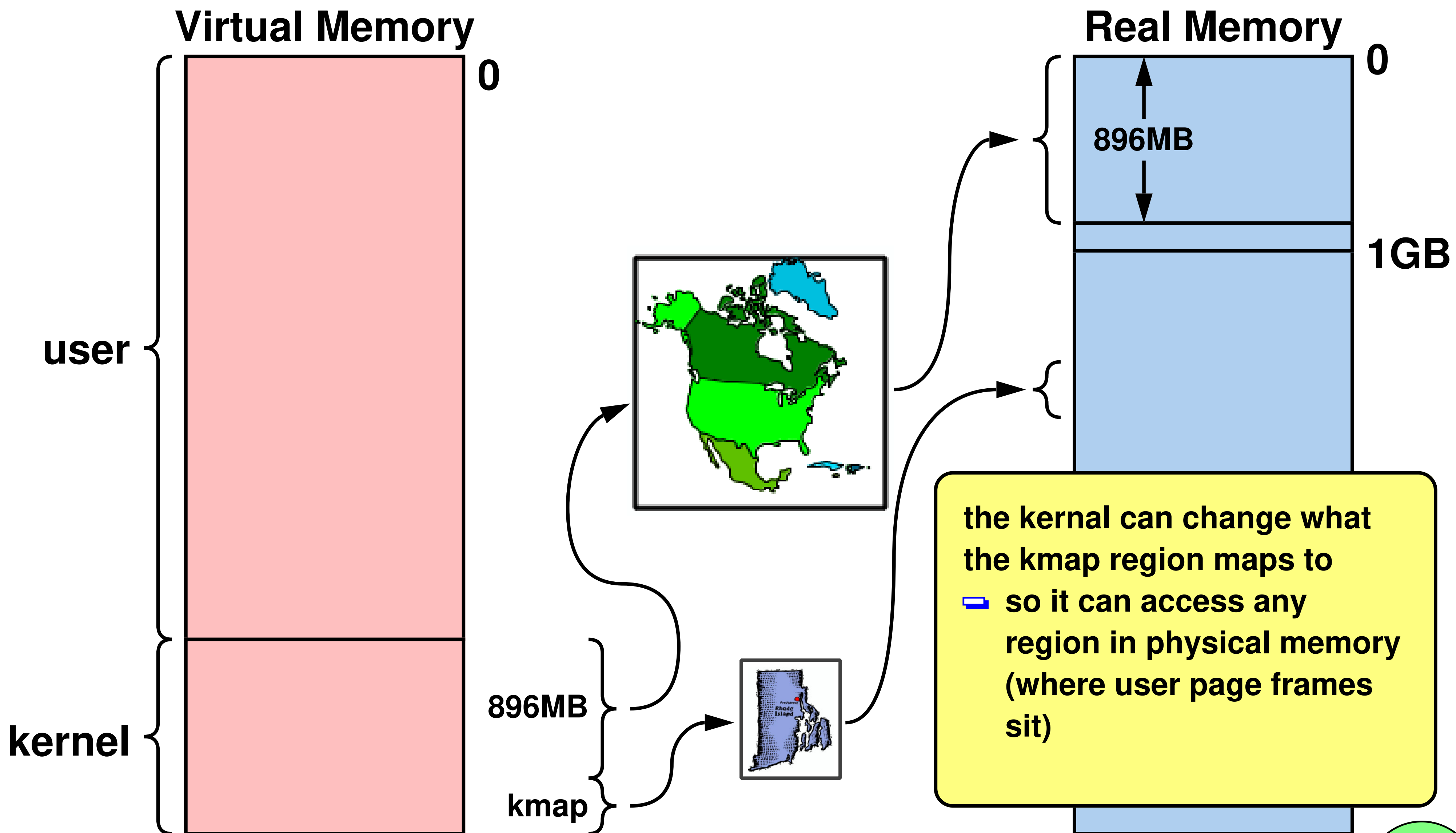


- In weenix, when an application call `read()` with buffer address `0x12345678`, how can the kernel write to this buffer?
  - should use *kernel* virtual address since it's always safe to use
  - how to convert `0x12345678` to kernel virtual address?
    - use the *vmmap* data structure
      - find memory segment it belongs (a memory segment consists of a bunch of page frames, find right page frame)
      - page frame has the base kernel virtual address (i.e., page-aligned)

Lots of Real Memory



Lots of Real Memory

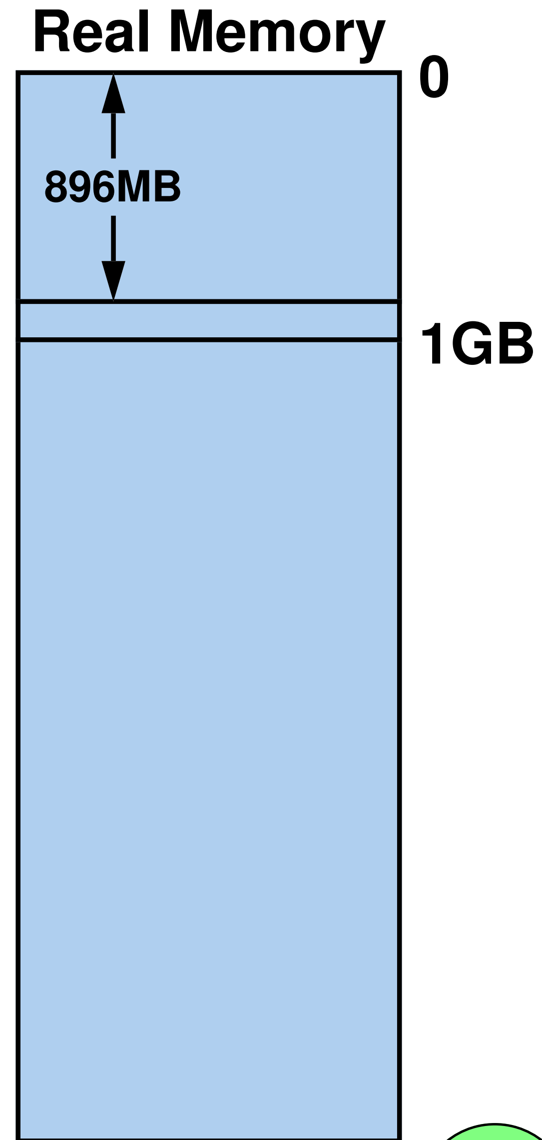


the kernal can change what the kmap region maps to

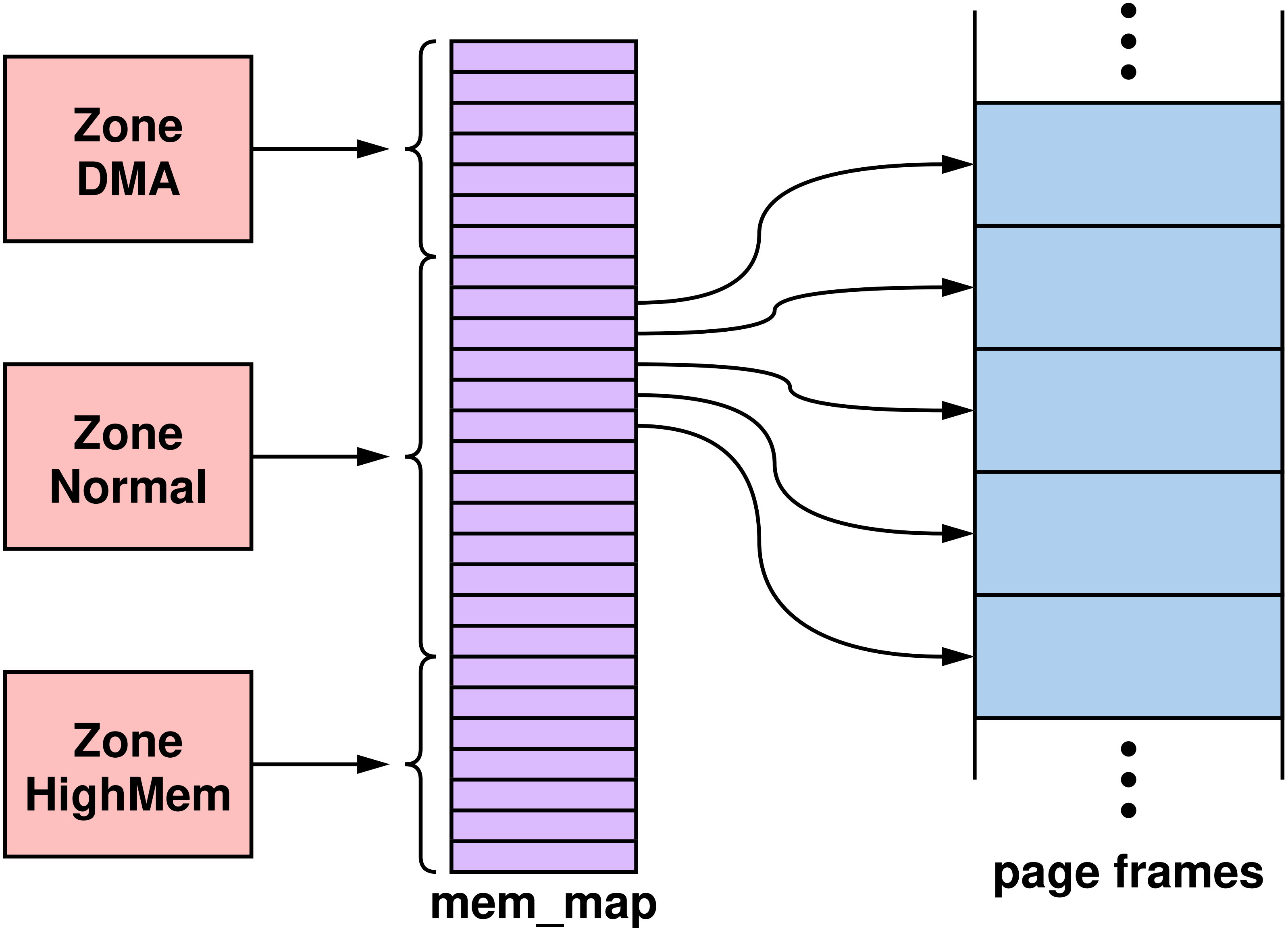
- so it can access any region in physical memory (where user page frames sit)

Mem_map and Zones

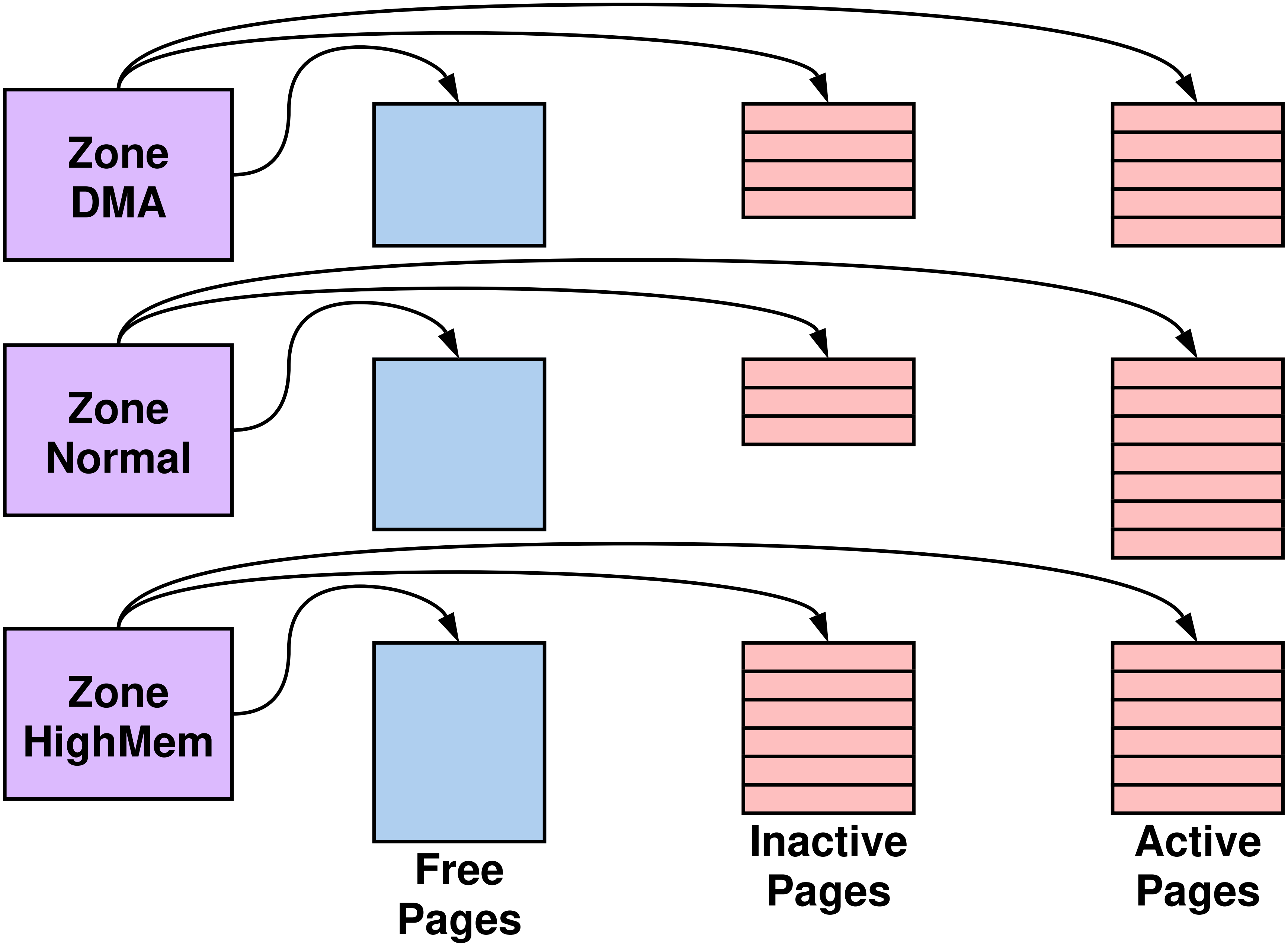
- Linux divides *physical memory* into 3 zones
  - *DMA zone:* locations $< 2^{24}$
    - `0x00000000` to `0x00ffffff`
    - many DMA devices can only handle 24-bit address
  - *Normal zone:* locations $\geq 2^{24}$ and $< 2^{30} - 2^{27}$
    - `0x01000000` to `0x37ffffff`
    - OS data structures must reside in this range
    - user pages *may* be in this range
  - *HighMem zone:* locations $\geq 2^{30} - 2^{27}$
    - `0x38000000+`
    - *strictly* for user pages



Mem_map and Zones



Page Lists



Zone
DMA
Zone
Normal
Zone
HighMem
Free
Pages
Inactive
Pages
Active
Pages

Page Lists

- Each zone's page frames are divided into three lists
  - *free list*
    - contain physical pages that have not been allocated
    - *buddy system* to maintain
  - *active*
    - picked out by *clock algorithm* as *recently used*
  - *inactive*
    - picked out by *clock algorithm* as *not recently used*
    - dirty/modified
      - marked as "*busy*" and is *unmapped* from all processes (i.e., set V=0 in PTE) that share this page
      - when you lookup a page frame in the page fault handler, if the page frame is "*busy*", you must wait until the disk operation is finished
      - when data transfer is completed, must wake up all threads waiting for this page frame to become "*un-busy*"