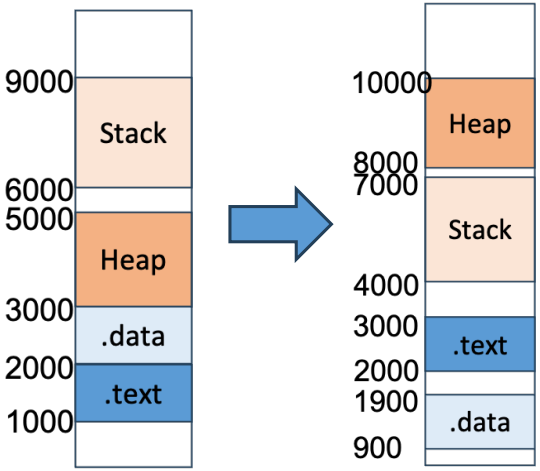
Lecture 17: Introduction to Paging

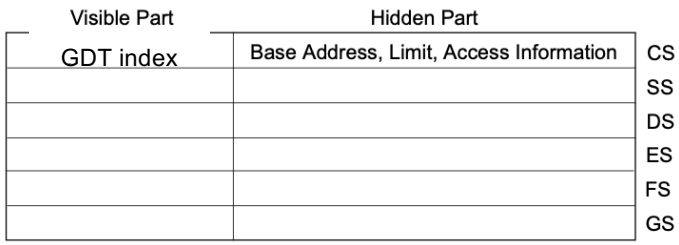
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture



Base	Bound	Type	Prv
2000	3000	RE	U
4000	7000	RW	U
900	1900	RW	U

Global Descriptor Table (GDT)



- Segment wise base/bound enables non-contiguous inter-segment memory
- VA has segment selector bits to identify the segment and an offset into that segment
- Base and bound for each segment stored in GDT
  - Address of GDT stored in GDTR
- x86 platform supports six segment registers per CPU
  - VA segment selector helps in choosing segment register
  - Each segment register contains the offset into GDT
  - Index into GDT is sum of GDT base address inside GDTR and above offset
  - Offset inside VA added to the segment's base address obtained from GDT
  - Base/bound is cached inside segment register's hidden part
- Segment registers must be saved/restored at context switch

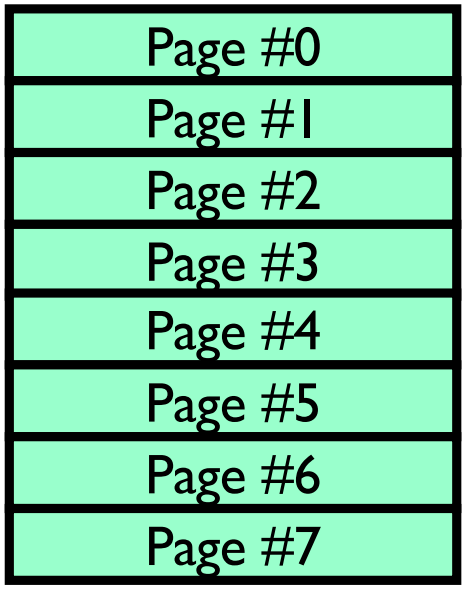
Today's Class

- Introduction to paging

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT

Paging



Physical Memory

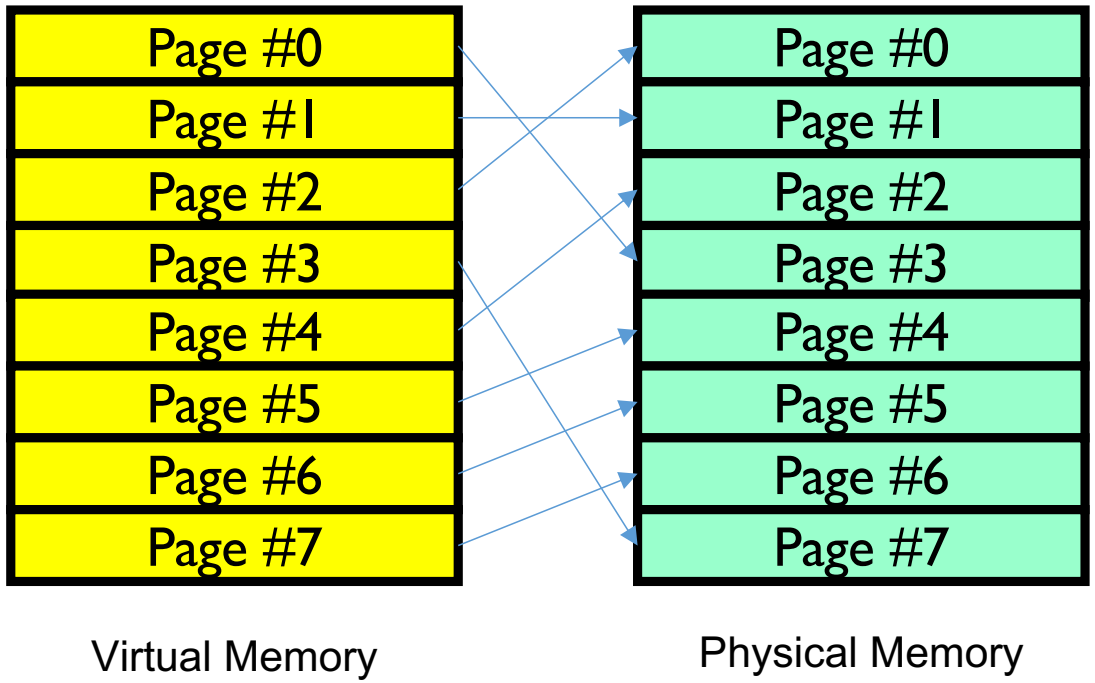
- Segmentation is easy to implement but it has major drawbacks because segments could have variable sized memory allocation
  - Can lead to **external** fragmentation
  - To allocate space for a new process, segments of some other process has to be swapped to the disk (huge overhead)
- Paging solves both the above issues by dividing the available memory into small and fixed size blocks (pages)
  - The memory can then be viewed as an array of fixed sized slots
  - For example, with a page size of 8 bytes, we can have an array of 8 pages on a DRAM whose size is 64 bytes

Total Number of Pages

- Default page size on Linux is 4096 bytes (4KB)
- Pages on physical memory depends on DRAM size
  - Assume DRAM size is 64GB, then total number of pages in **physical memory** will be 64GB/4KB, i.e., 16 x 10^6 pages
- Pages in virtual memory is fixed as per the addressing mode
  - Total number of pages in **32-bit** addressing mode is 2^{32}/4KB, i.e., 2^{20}
  - Total number of pages in **64-bit** addressing mode is 2^{64}/4KB, i.e., 2^{52}

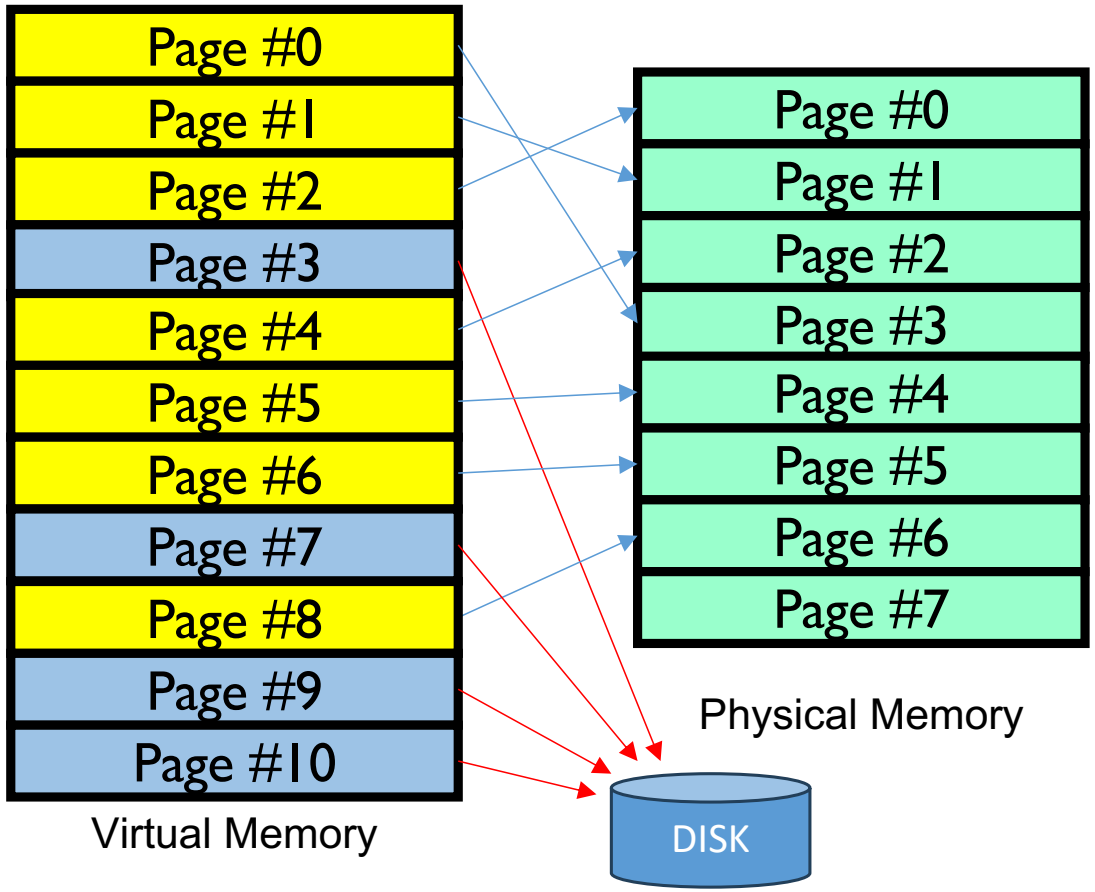
If you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!) -- OSTEP Book

Paging



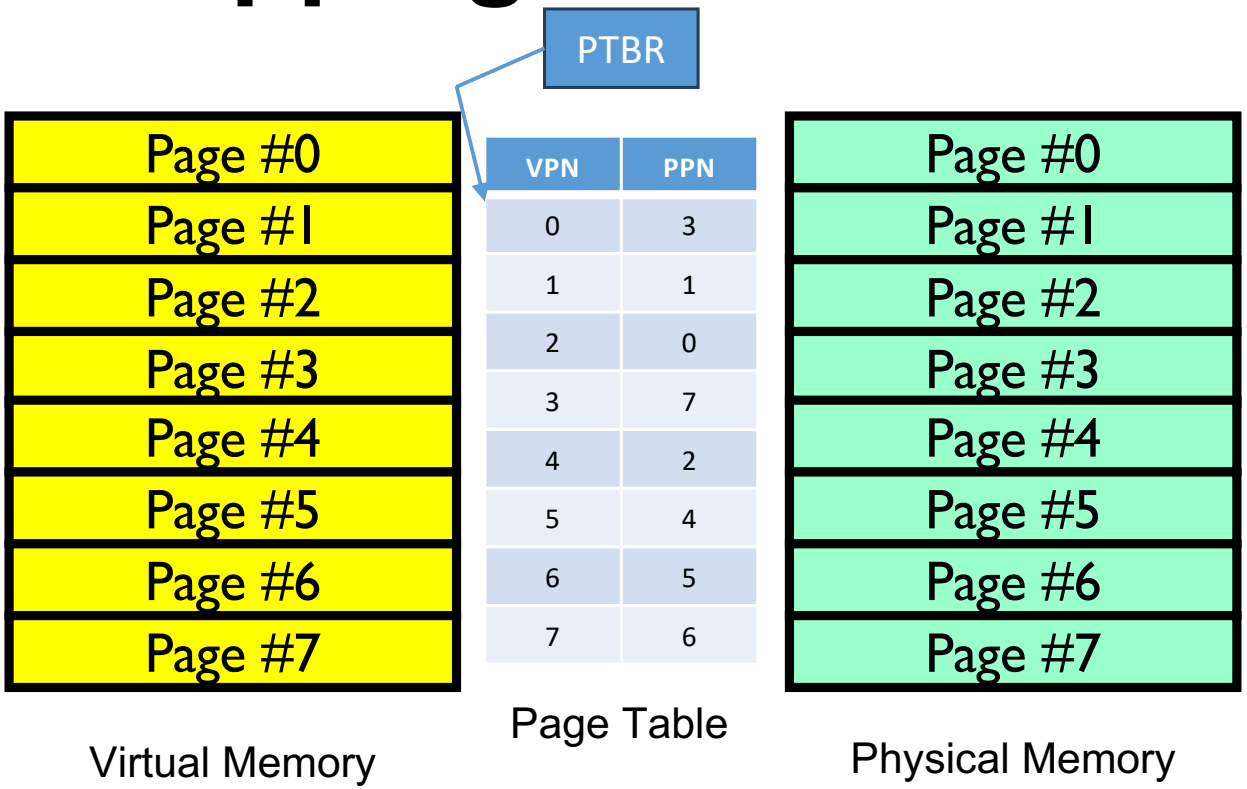
- Process's segments would still be of variable size, but now instead of a contiguous chunk of memory, they could have **non-contiguous** mapping with physical pages (both inter/intra-segment)
- How about internal fragmentation?
  - Yes, if a segment requires 5000 bytes of memory, it would have 2 page allocations where 4092 bytes would forever remain free in the second page

Paging



- Swapping out to disk can still happen, but where would the overheads will be minimal?
  - Segmentation Or paging?
- Lesser overheads in paging as compared to segmentation
  - Instead of swapping out an entire segment, only unused pages of program segment(s) will be swapped out
- Issue still remaining
  - How to map Virtual Page Number (VPN) to Physical Page Number (PPN)?

Mapping VPN to PPN



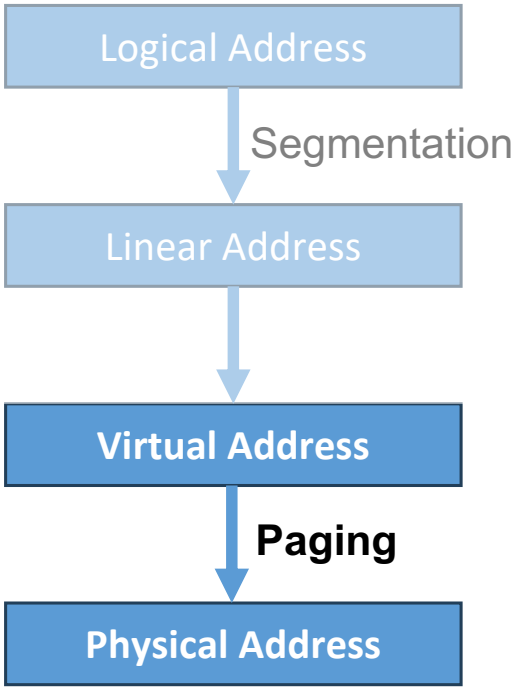
- Mapping of VPN to PPN is carried out using a data structure called as Page Table
- It is stored in DRAM
- Each CPU has a Page Table Base Register (PTBR) that holds the base address and Page Table Length Register (PTLR) that holds the size of the page table

A System Using Paging

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_pagetable_base_registers(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Page table resides in memory
- Process's PCB has a field to store PTBR
- PTBR of a running process is saved during context switch

Segmentation Along with Paging

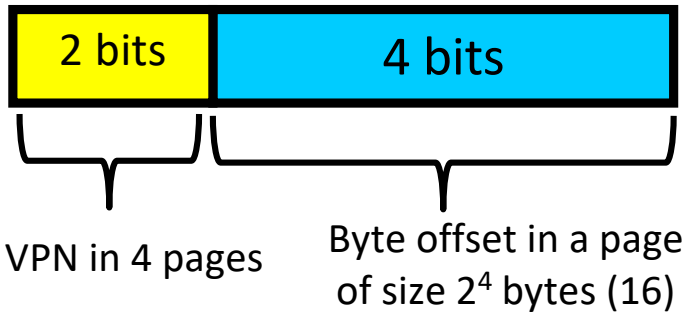


- Segmentation cannot be disabled in x86 processors, but address translation is complex using the segmentation-based approach
  - Hence, to simplify the address translation, segment registers are set to have the same base (0) and bound (2^{32}-1) values (Lecture #16)
- The address visible to a programmer is actually called as Logical address, as it has an associated segment selector (Lecture #16)
  - Segmentation converts this logical address to linear address, which is then converted into virtual address to support paging
- However, as the base and bound of each segment registers maps to entire 2^{32} memory size (4GB), logical address will be the same as virtual address
- Hence, we will call the address visible to the programmer as a virtual address in all the remaining slides/lectures

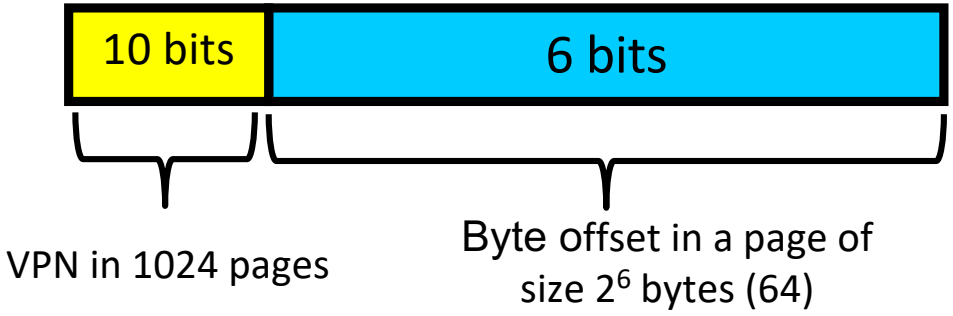
Encoding VPN inside Virtual Address

- How many VAs can reside in a 4KB page size?
  - 4096 bytes/4 bytes = 1024 virtual addresses
- How to find the VPN for a given VA?
  - Encode (prefix) the VPN inside each VA (**next slide**)
- How many bits required to represent a VPN for pages of N bytes in a virtual memory of M bytes (total M/N pages)?
  - M=64 (2^6) and N=16
    - Pages = 4 and bits required to identify 4 pages are 2 (2^2=4)
  - M=4GB (2^{32}) and N=4KB
    - Pages = $2^{32}/4096 = 2^{20}$ and bits required to identify 2^{20} pages are 20

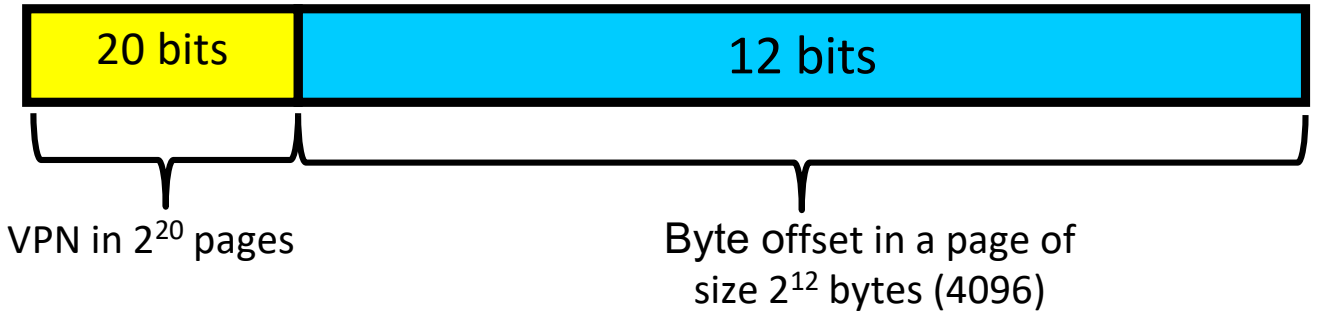
Two Components of a Virtual Address



6-bit addressing with page size of 16 bytes

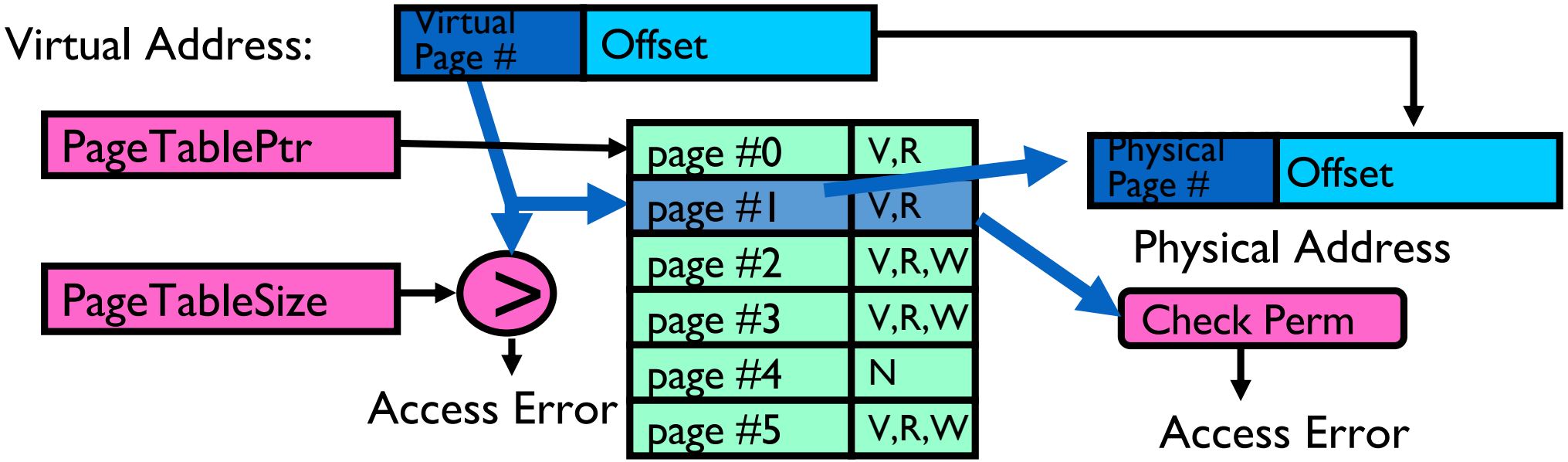


16-bit addressing with page size of 64 bytes



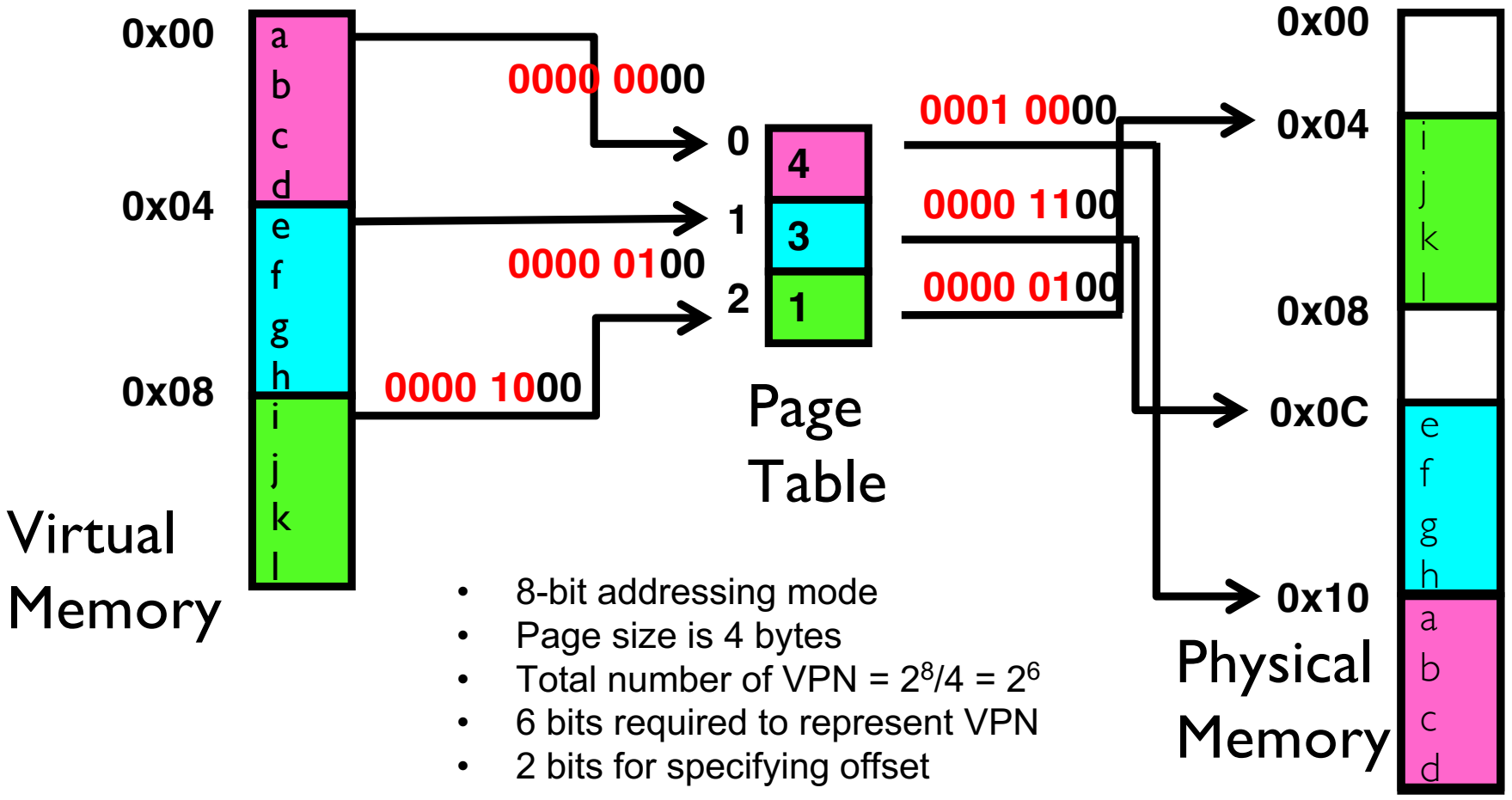
32-bit addressing with page size of 4KB

How to Implement Simple Paging?



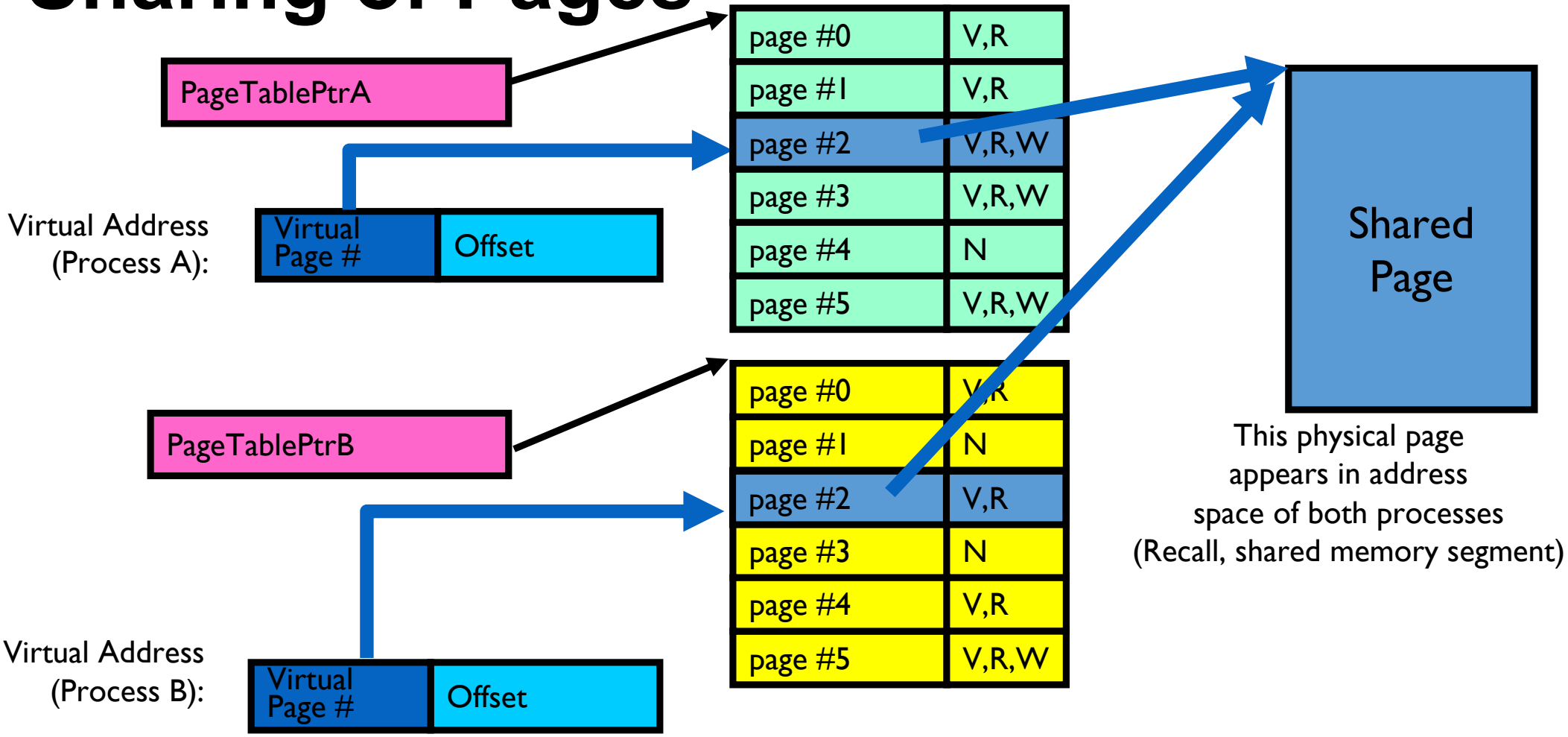
- Page Table (One per process and resides in physical memory)
  - Contains physical page and permission for each virtual page (e.g. **V**alid bits, **R**ead, **W**rite, etc)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
  - Virtual page # is all remaining bits (Physical page # copied from table into physical address)
  - Check Page Table bounds and permissions

A Simple Page Table in Action



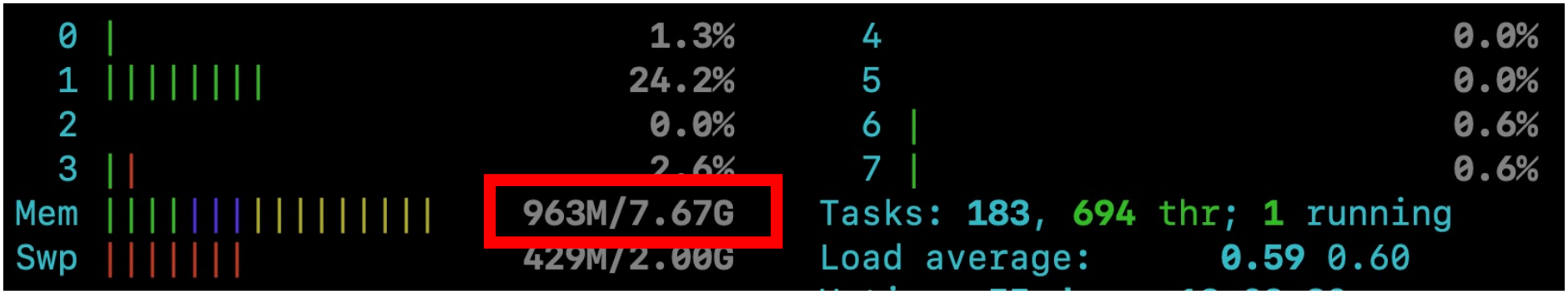
- 8-bit addressing mode
- Page size is 4 bytes
- Total number of VPN = $2^8/4 = 2^6$
- 6 bits required to represent VPN
- 2 bits for specifying offset

Sharing of Pages



Physical Page Allocation in Linux

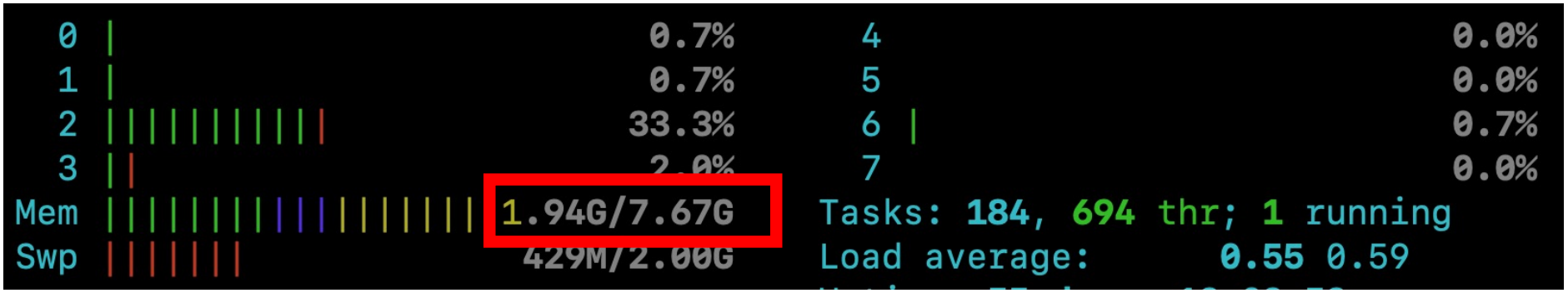
```
char * array = (char*) malloc(sizeof(char)*1024*1024*1024);
```



- The above program allocates 1GB memory using malloc, but output of **htop** command doesn't show 1GB memory usage!
  - I did not see any change to that memory even after the above program finished, which implies its default memory usage in my system
    - What is happening?

Physical Page Allocation in Linux

```
char * array = (char*) malloc(sizeof(char)*1024*1024*1024);
memset(array, 0x00, sizeof(char)*1024*1024*1024);
```

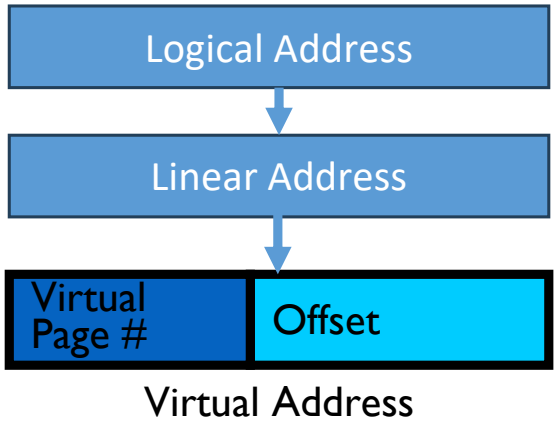


- Just writing to that allocated space changes the output of **htop** command to show the 1.94GB memory usage (1GB from the program and 0.94GB of default memory)
  - What just happened now?

Physical Page Allocation in Linux

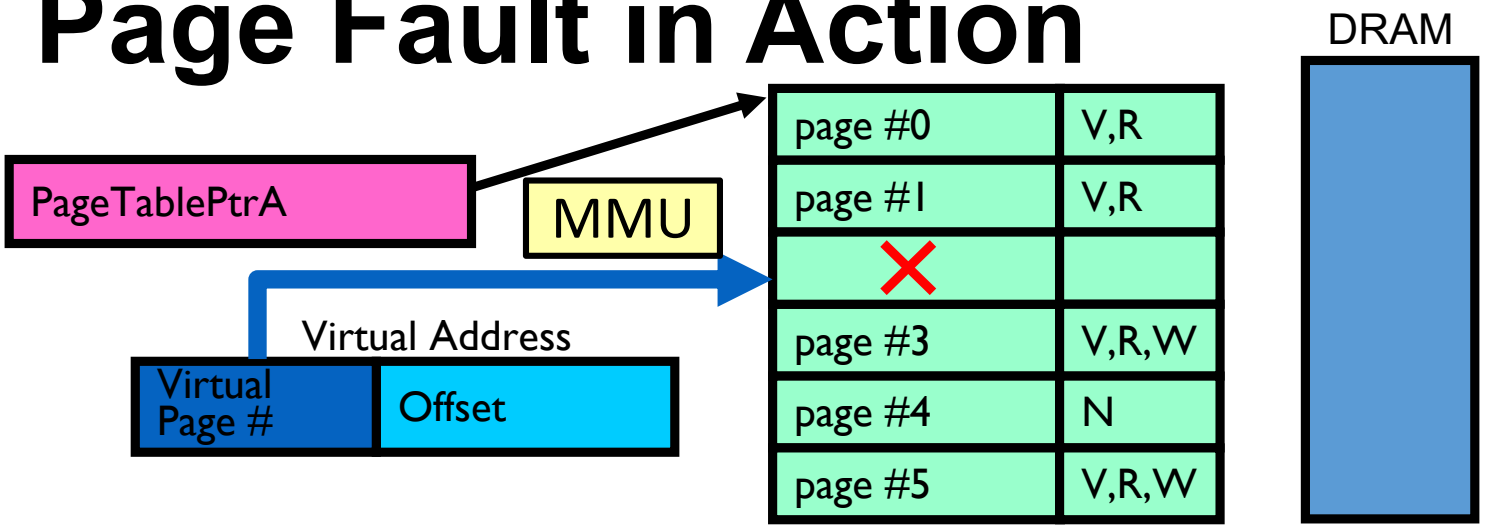
- Linux uses lazy memory allocation policy to save memory space (RAM is precious!)
  - Not just for malloc, but even while loading program segments
  - Physical memory for the segments are not allocated until accessed
    - E.g., if a program has a .data segment, but the user doesn't accesses the global variables (residing in .data segment) then it doesn't make sense to allocate memory for .data segment upfront (load time)
- So, how the physical memory is allocated?
  - During the program execution, when the program attempts to **read/write** to that memory
    - Linux uses a mechanism known as **Page Fault** for lazy page allocation

Page Fault in Action



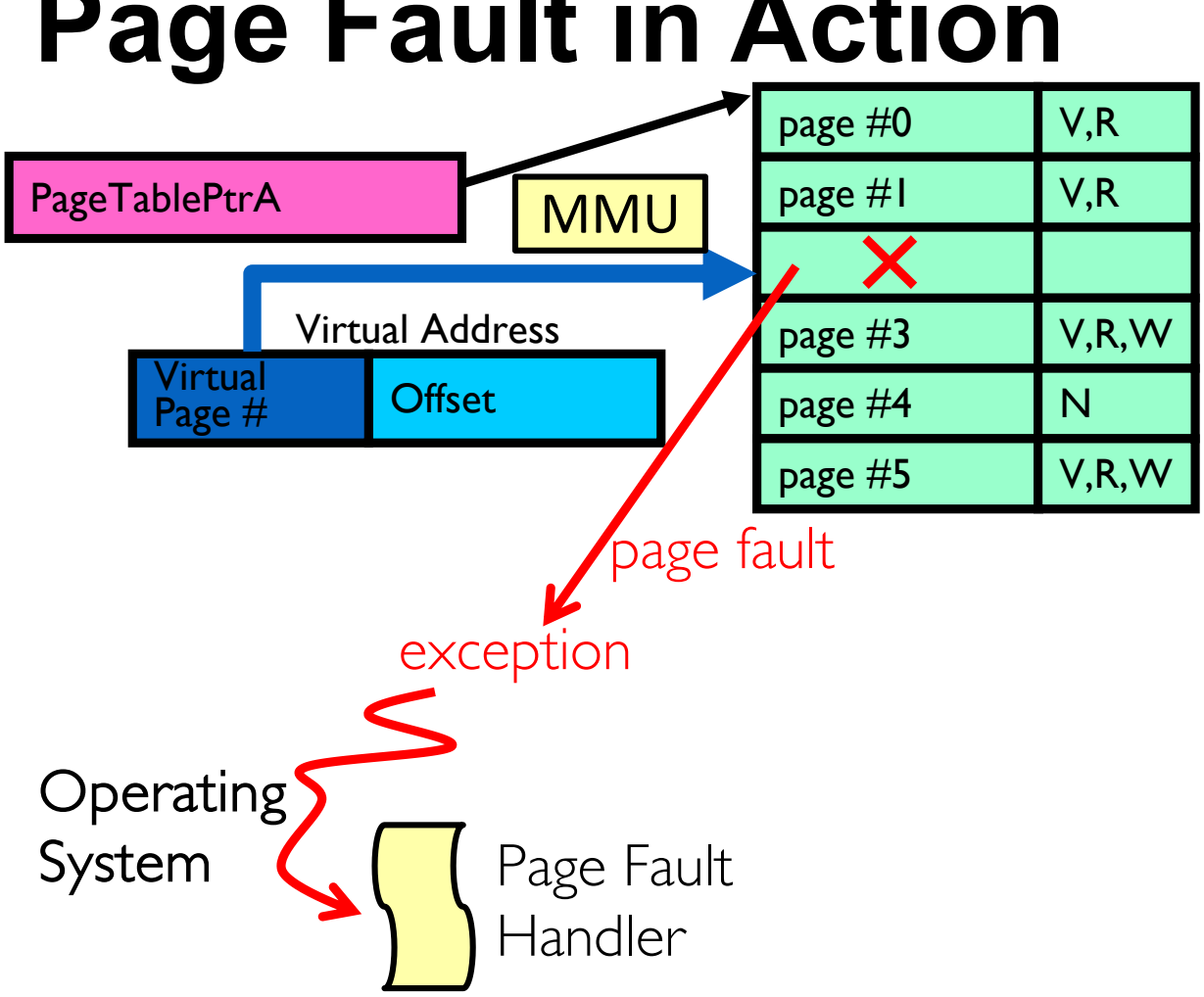
- Program attempts to access an address
- Segmentation converts this logical address to linear address, which is then converted into virtual address to support paging

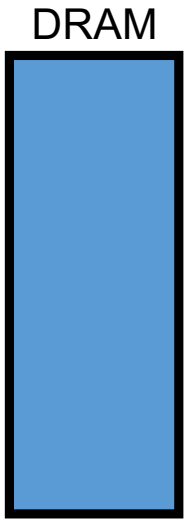
Page Fault in Action



- This address is valid (i.e., it points to a valid heap allocation)
- However, the page table doesn't have any entry for this VA to PA mapping

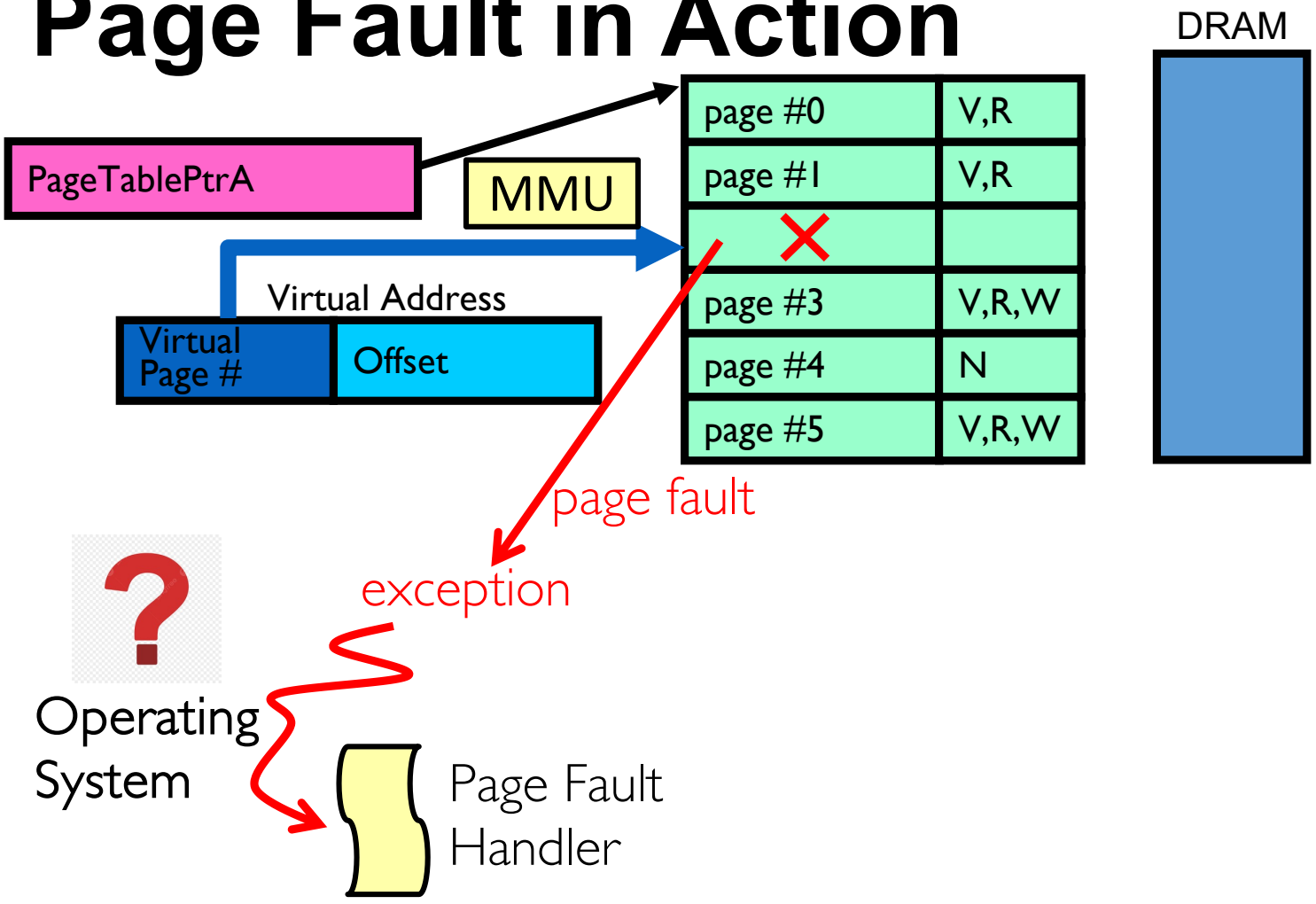
Page Fault in Action





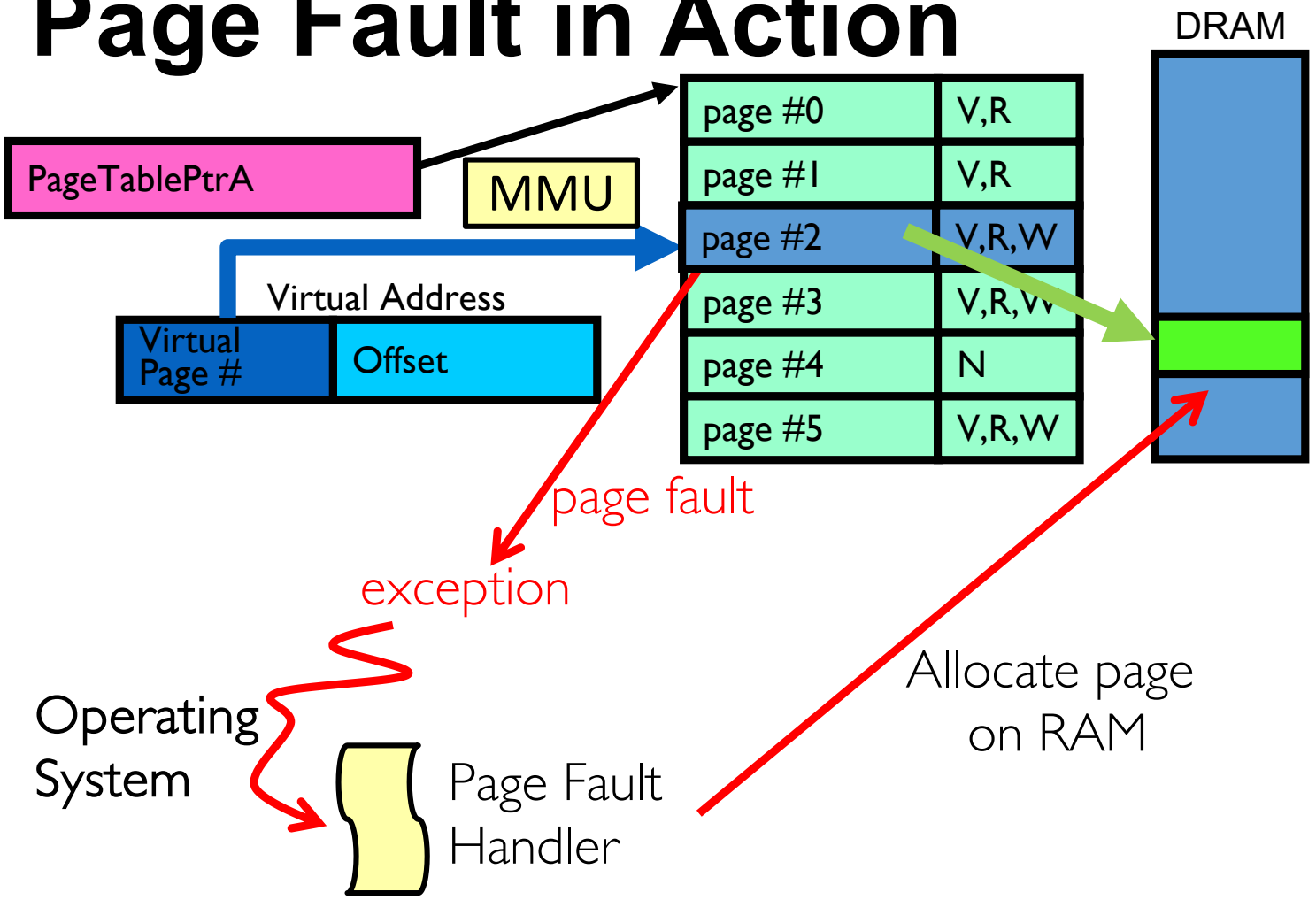
- Program state is saved and Interrupt 14 is generated by the MMU indicating page fault exception

Page Fault in Action



- OS will decide what action is to be taken
  - It could be that the page was allocated but swapped out to disk (i.e., demand paging, we will see in future lecture)
- OS finds out that no allocation was ever made

Page Fault in Action



- OS will allocate page(s) on DRAM
- Page table entry is updated
- Program execution (state) is restored from the same location that caused the page fault

Reference

- Chapter 18 in OSTEP book (“Paging Introduction”)

Next Lecture

- Translation Lookaside Buffer (TLB)
- Quiz-3 during next lecture
  - Syllabus: Lectures 13, 15-17