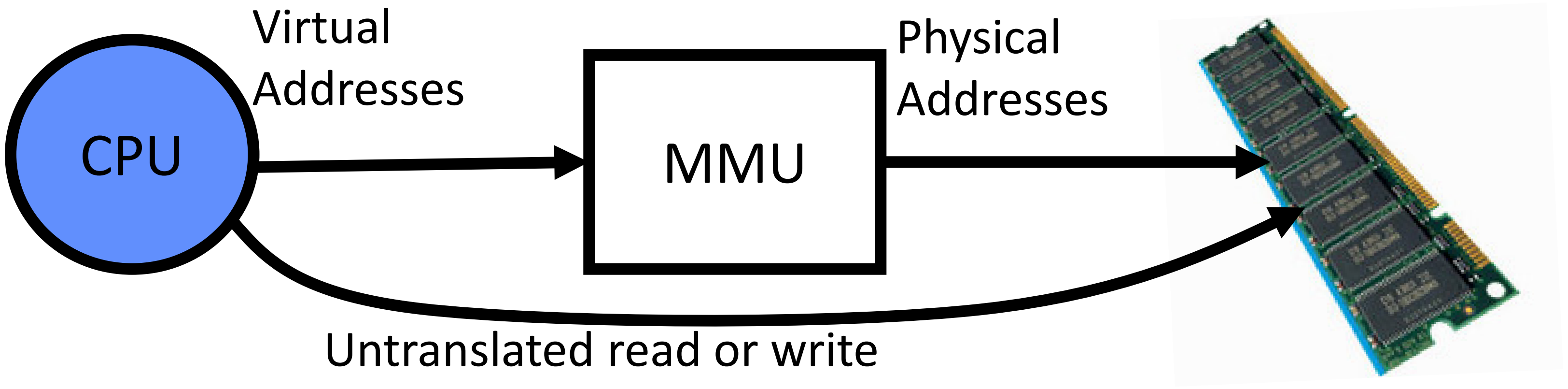
CSC 112: Computer Operating Systems
Lecture 14

Memory 2: Virtual Memory (Con’t), Caching and TLBs

Department of Computer Science,
Hofstra University

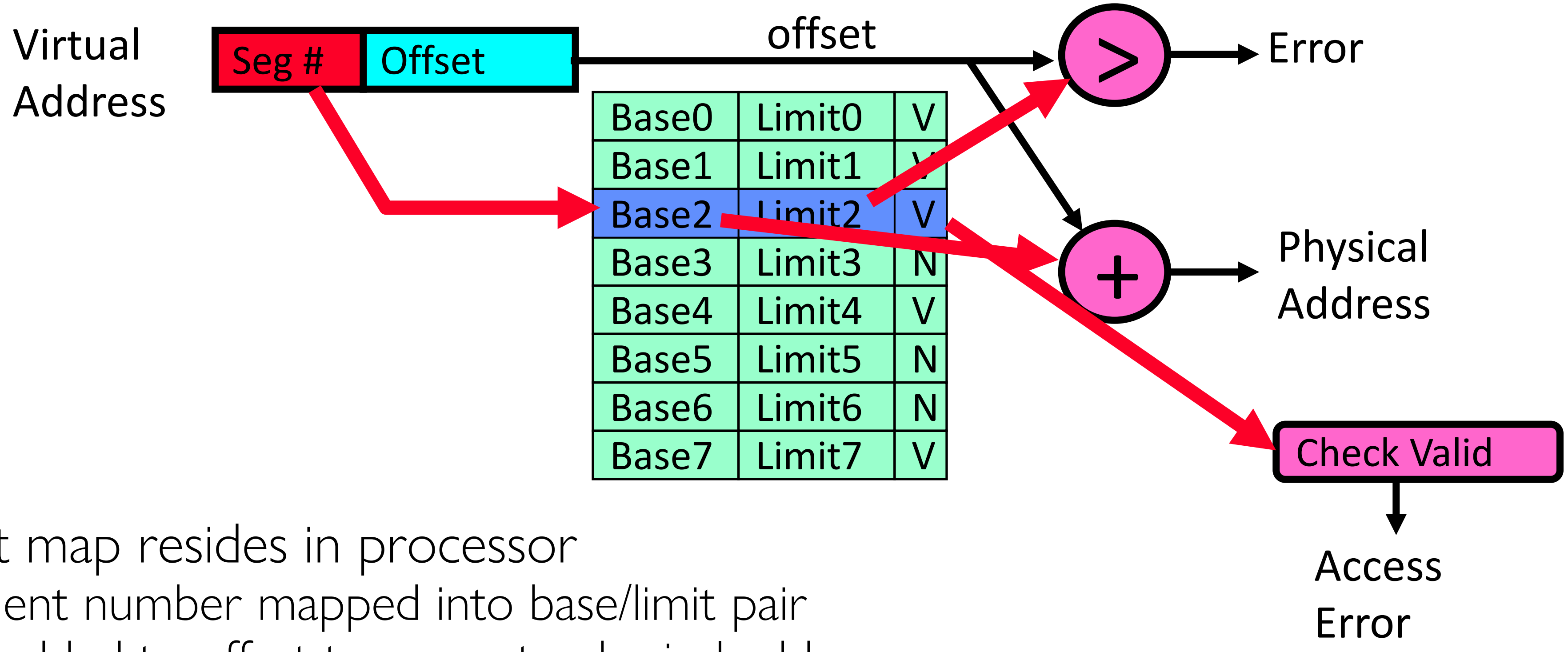
Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

Recall: General Address translation



- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between the two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
  - Extra benefit: every program can be linked/loaded into same region of user address space

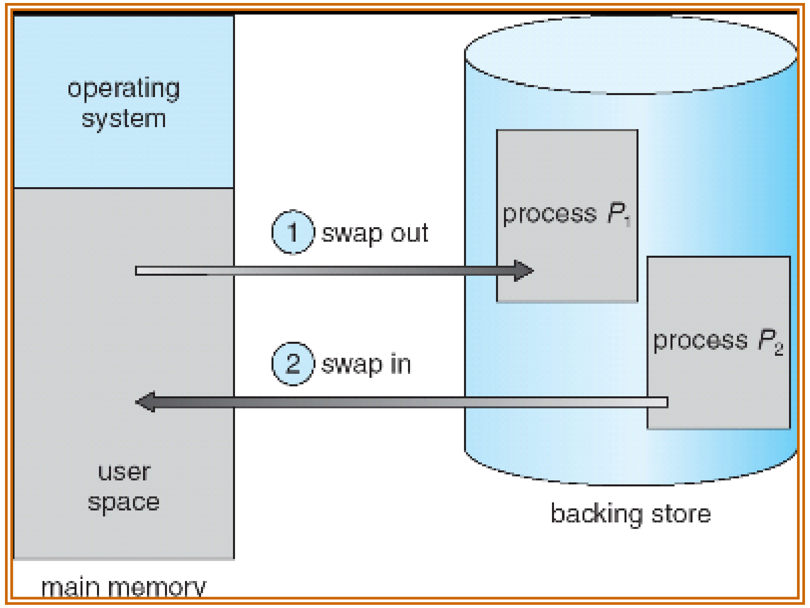
Recall: Multi-Segment Model



Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: mov [es:bx],ax.
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well

What if not all segments fit in memory?



- Extreme form of Context Switch: Swapping
  - To make room for next process, some or all of the previous process is moved to disk
    - » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

Problems with Segmentation

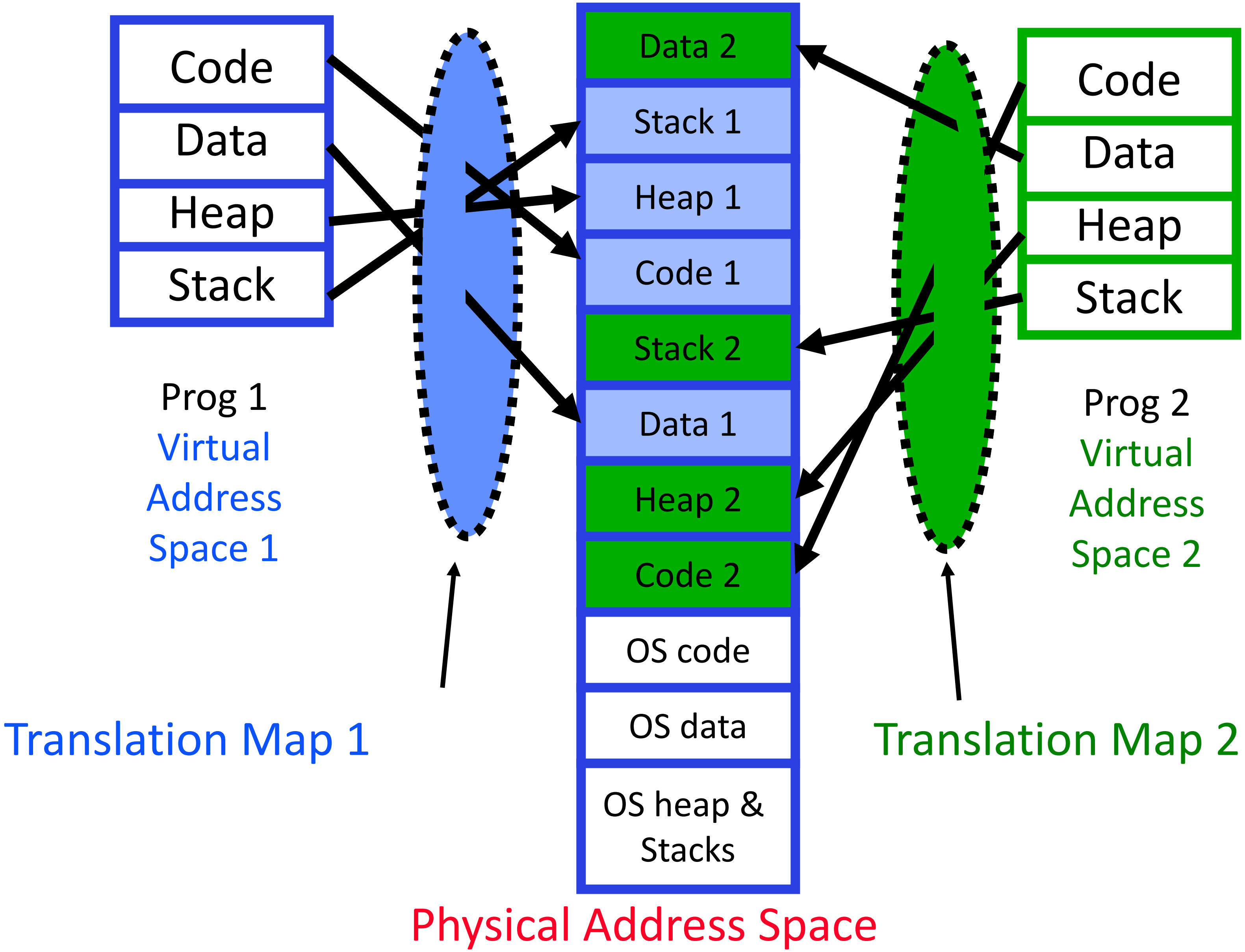
- Must fit variable-sized chunks into physical memory

- May move processes multiple times to fit everything

- Limited options for swapping to disk

- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks

Recall: General Address Translation

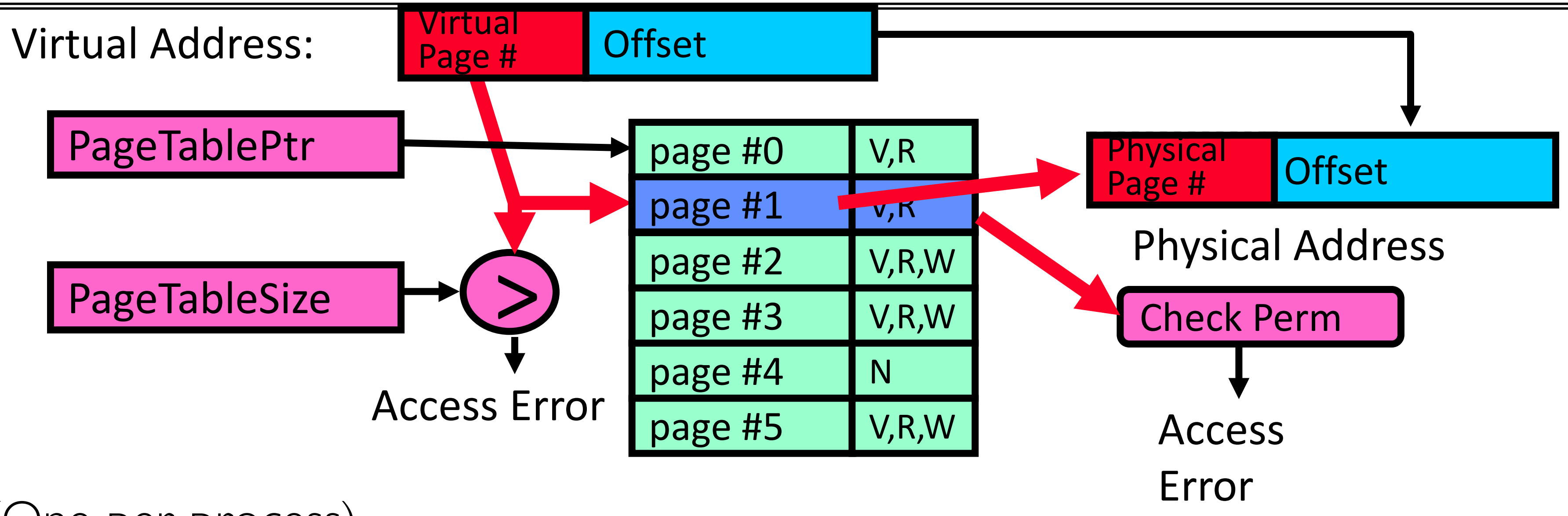


Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      `00110001110001101 … 110010`
    - » Each bit represents page of physical memory
      1 ⇒ allocated, **0** ⇒ free

- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

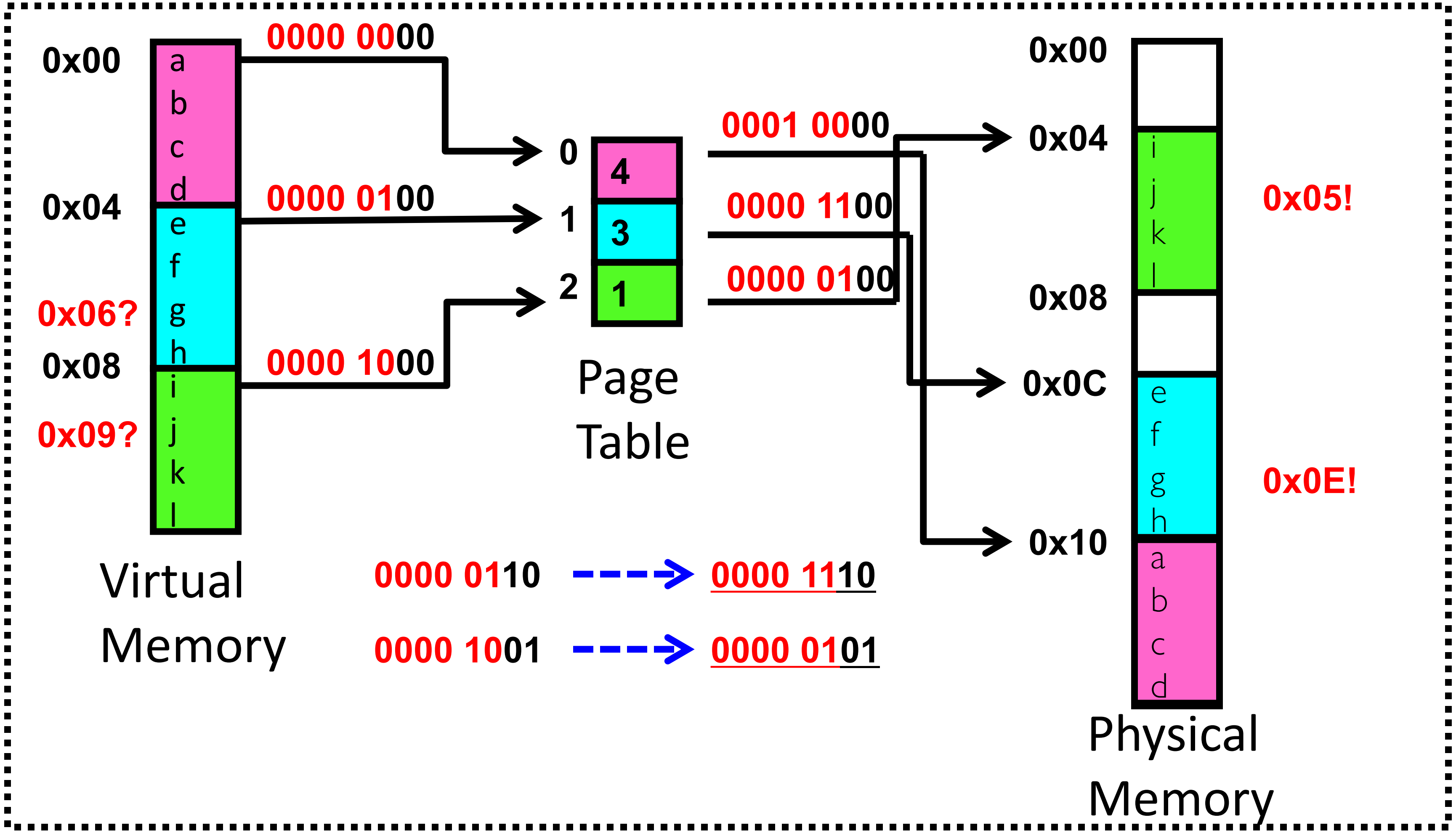
How to Implement Simple Paging?



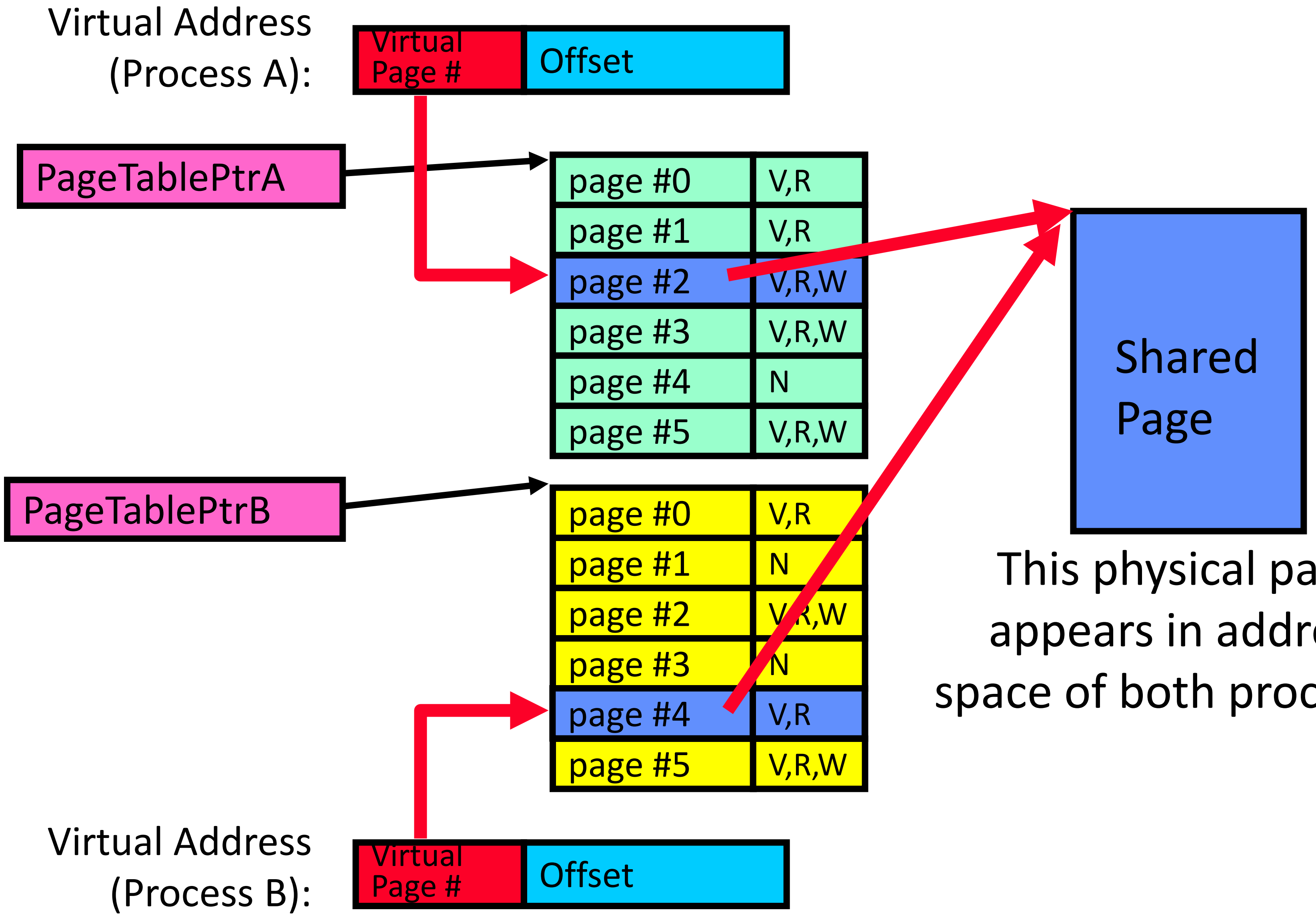
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

Simple Page Table Example

Example (4 byte pages)



What about Sharing?



page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

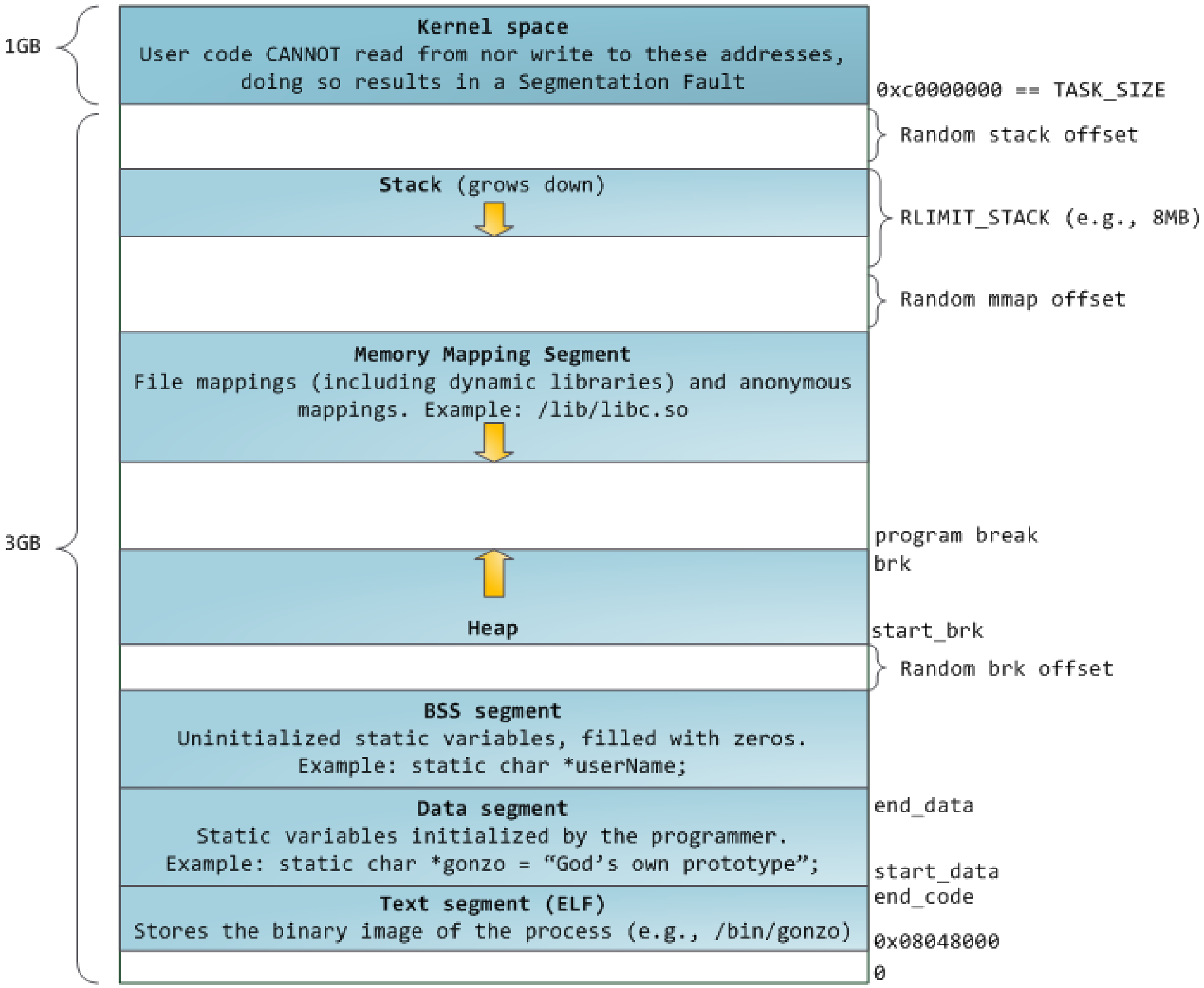
page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

This physical page appears in address space of both processes

Where is page sharing used ?

- The “kernel region” of every process has the same page table entries
  - The process cannot access it at user level
  - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    - What does the kernel need to do to access other user processes?
- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  - Can actually share objects directly between processes
    - Must map page into same place in address space!
  - This is a limited form of the sharing that threads have within a single process

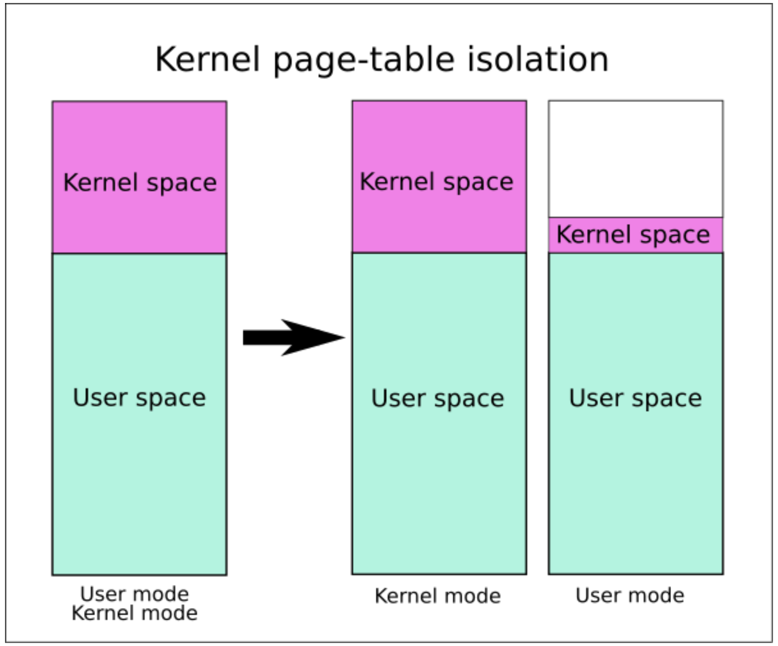
Recall: Memory Layout for Linux 32-bit (Pre-Meltdown patch!)


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

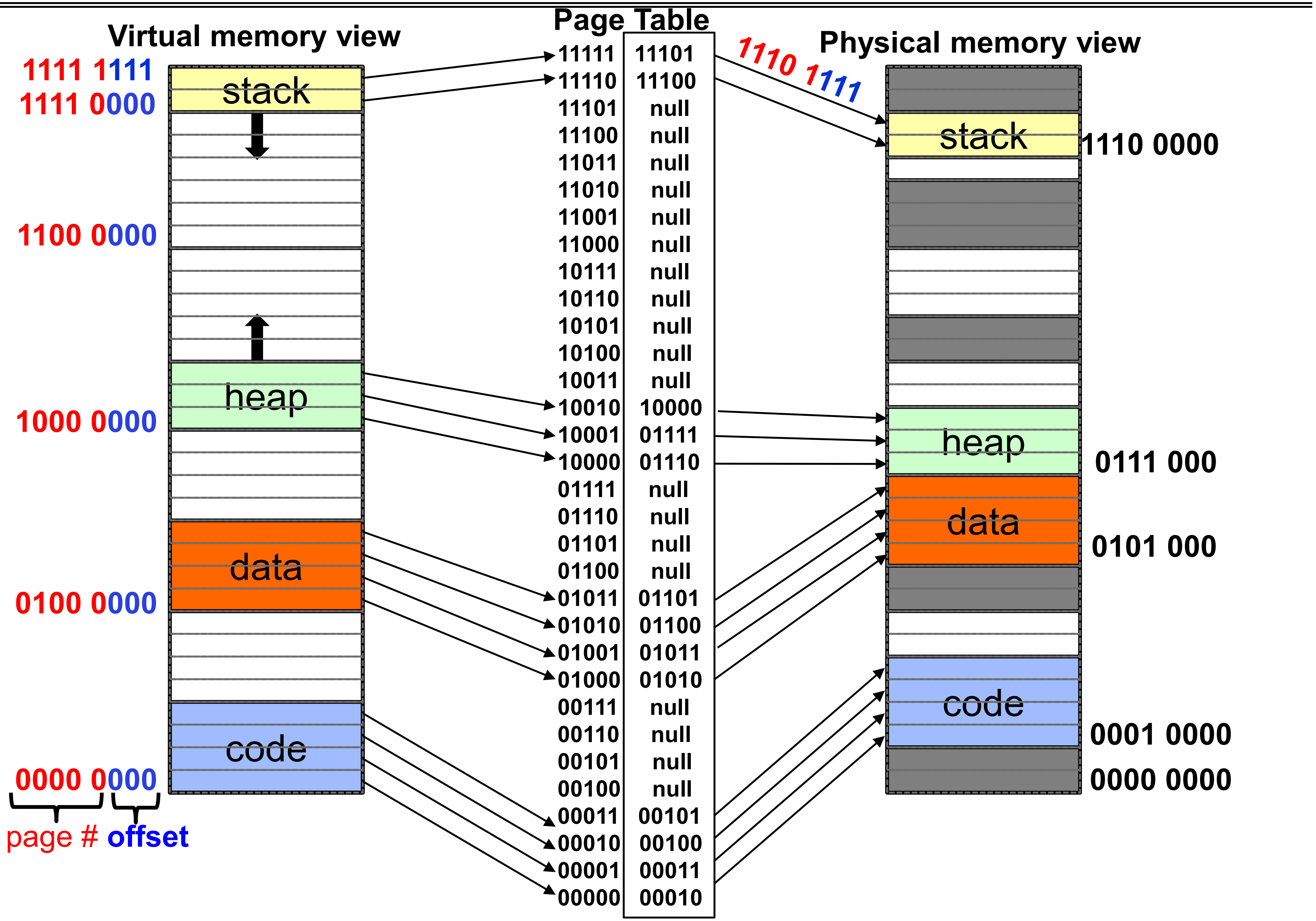
http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Some simple security measures

- Address Space Randomization
  - Position-Independent Code ⇒ can place user code anywhere in address space
    - » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don't map whole kernel space into each process, switch to kernel page table
  - Meltdown⇒map none of kernel into user mode!



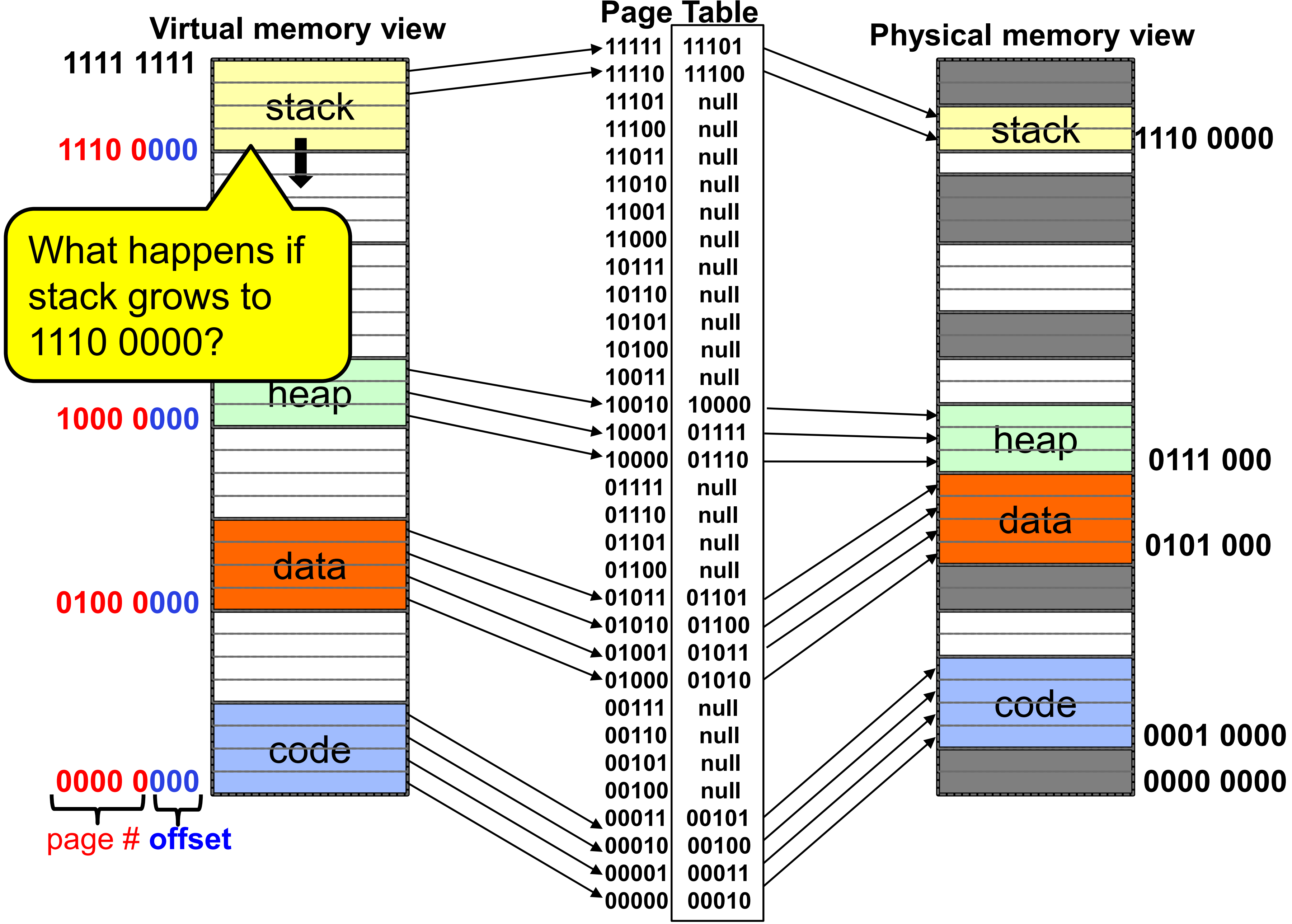
Summary: Paging



Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Summary: Paging



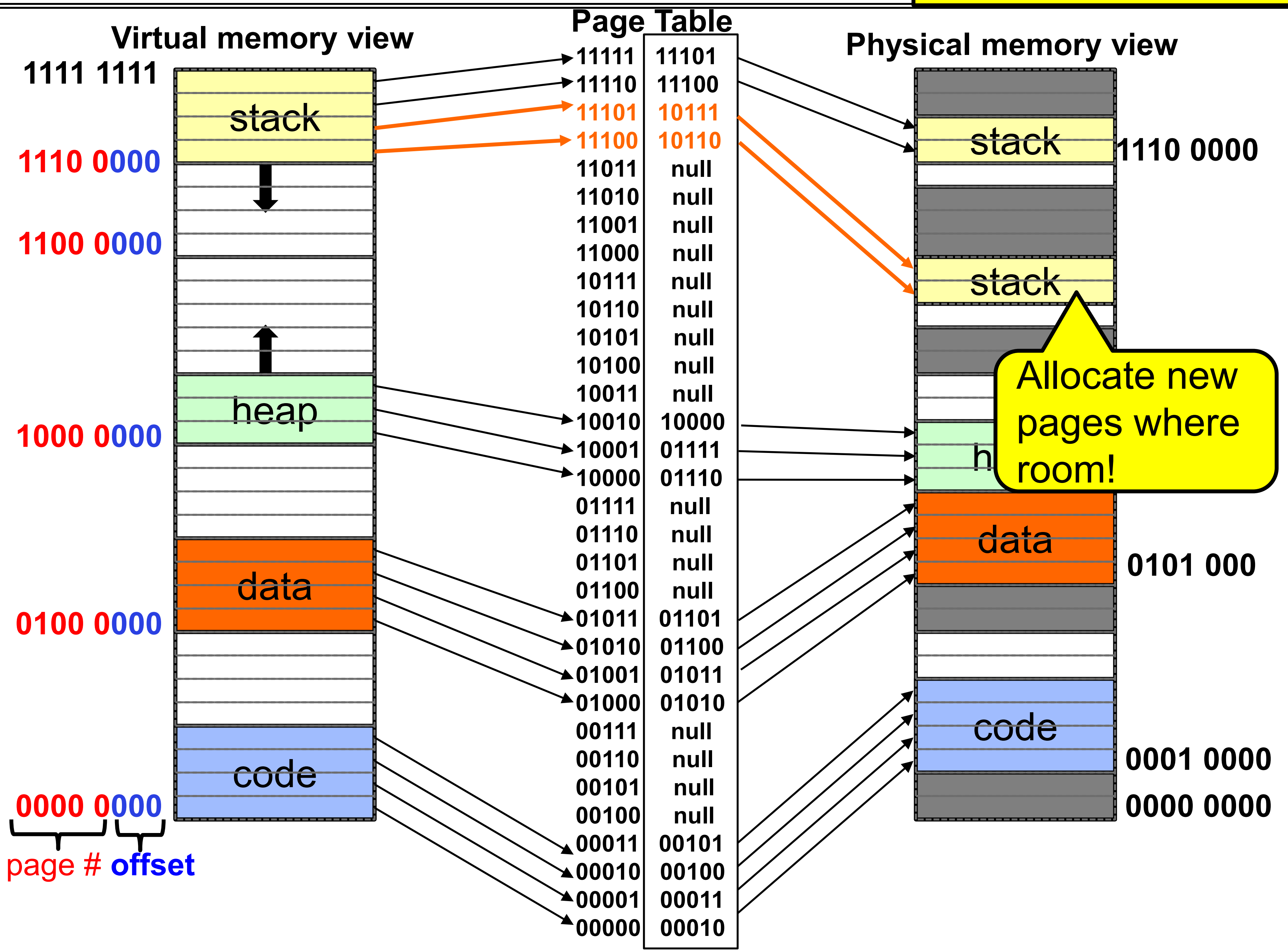
What happens if stack grows to 1110 0000?

Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Summary: Paging

Challenge: Table size equal to # of pages in virtual memory!



Allocate new pages where room!

Page Table

11111	11101
11110	11100
11101	10111
11100	10110
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

How big do things get?

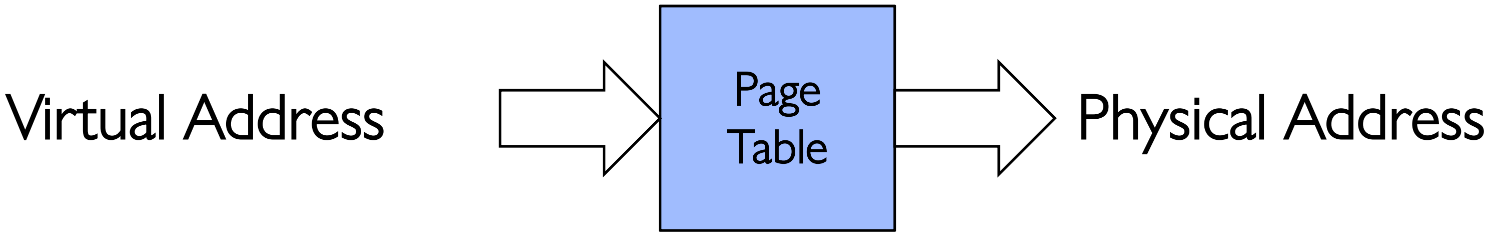
- 32-bit address space => 2^{32} bytes (4 GB)
  - Note: "b" = bit, and "B" = byte
  - And *for memory*:
    - » "K"(kilo) = 2^{10} = 1024 ≈ 10^3 (But not quite!): Sometimes called "Ki" (Kibi)
    - » "M"(mega) = 2^{20} = $(1024)^2$ = 1,048,576 ≈ 10^6 (But not quite!): Sometimes called "Mi" (Mibi)
    - » "G"(giga) = 2^{30} = $(1024)^3$ = 1,073,741,824 ≈ 10^9 (But not quite!): Sometimes called "Gi" (Gibi)
- Typical page size: 4 KB
  - how many bits of the address is that ? (remember 2^{10} = 1024)
  - Ans – 4KB = $4\times2^{10} = 2^{12} \Rightarrow$ 12 bits of the address
- So how big is the simple page table for *each* process?
  - $2^{32}/2^{12} = 2^{20}$ (that's about a million entries) x 4 bytes each => 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory
- How big is a simple page table on a 64-bit processor (x86_64)?
  - $2^{64}/2^{12} = 2^{52}$(that's 4.5×10^{15} or 4.5 exa-entries)×8 bytes each = 36×10^{15} bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
  - This is really a lot of space – for only the page table!!!
- The address space is *sparse*, i.e. has holes that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing

Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) *and* dual-mode!
  - Can't let process alter its own page table!
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to share
  - Con: What if address space is sparse?
    - » E.g., on UNIX, code starts at 0, stack starts at (2^{31}-1)
    - » With 1K pages, need 2 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation

How to Structure a Page Table

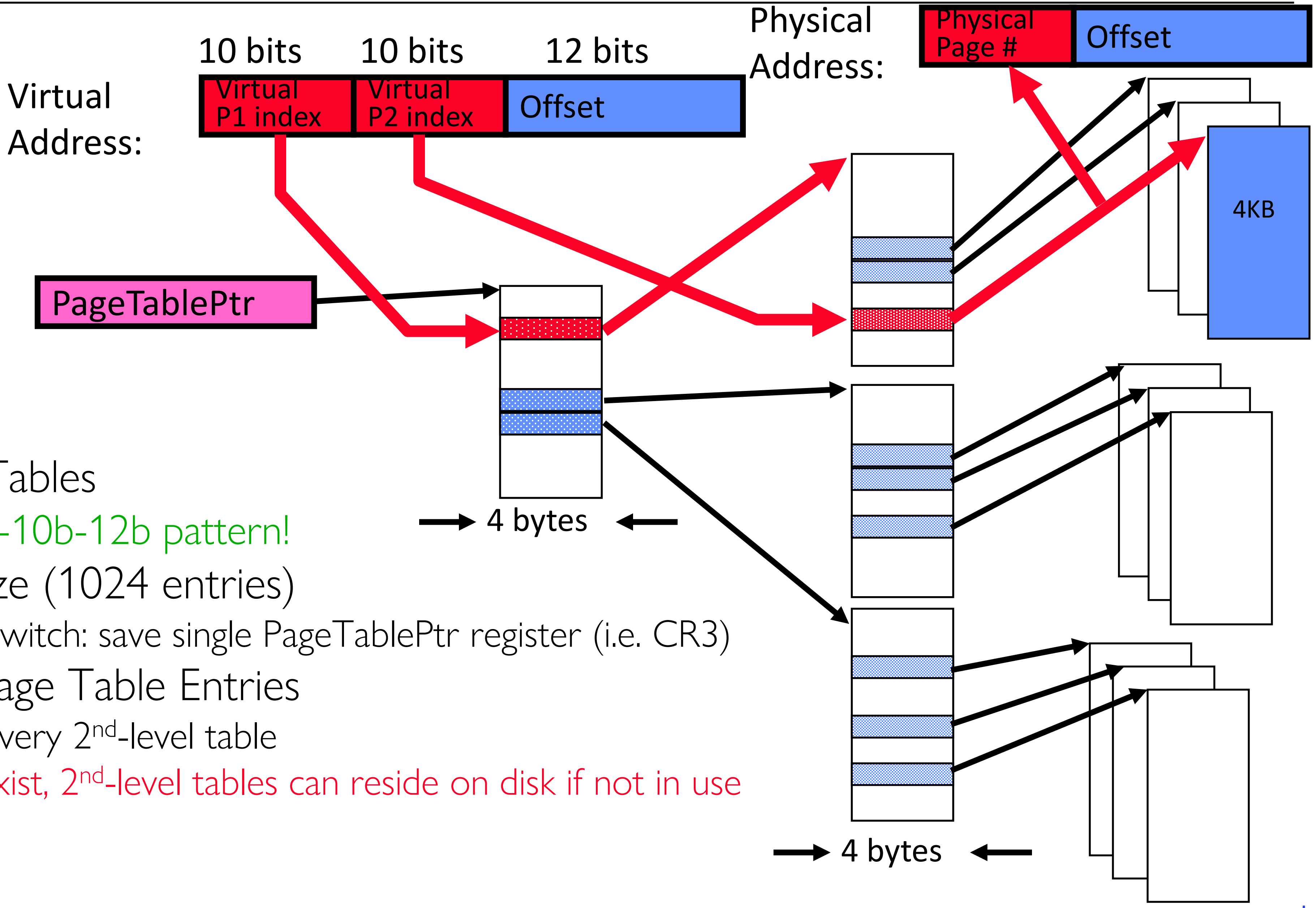
- Page Table is a *map* (function) from VPN to PPN



- Simple page table corresponds to a *very large* lookup table
  - VPN is index into table, each entry contains PPN

- What other map structures can you think of?
  - Trees?
  - Hash Tables?

Fix for sparse address space: The two-level page table



- Tree of Page Tables
  - “Magic” 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don’t need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Example: x86 classic 32-bit address translation

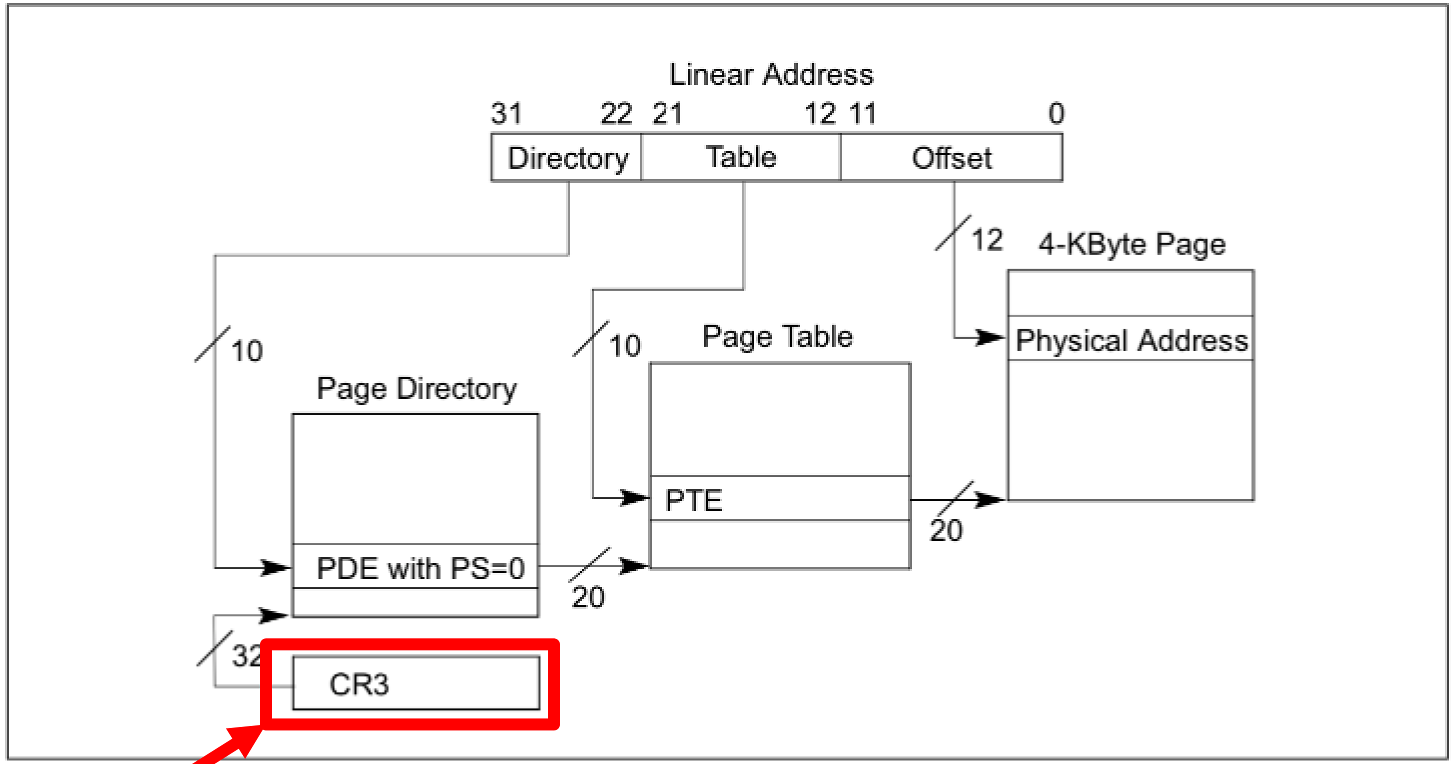


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

- Intel terminology: Top-level page-table called a "Page Directory"
  - With "Page Directory Entries"
- CR3 provides physical address of the page directory
  - This is what we have called the "PageTablePtr" in previous slides
  - Change in CR3 changes the whole translation table!

What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

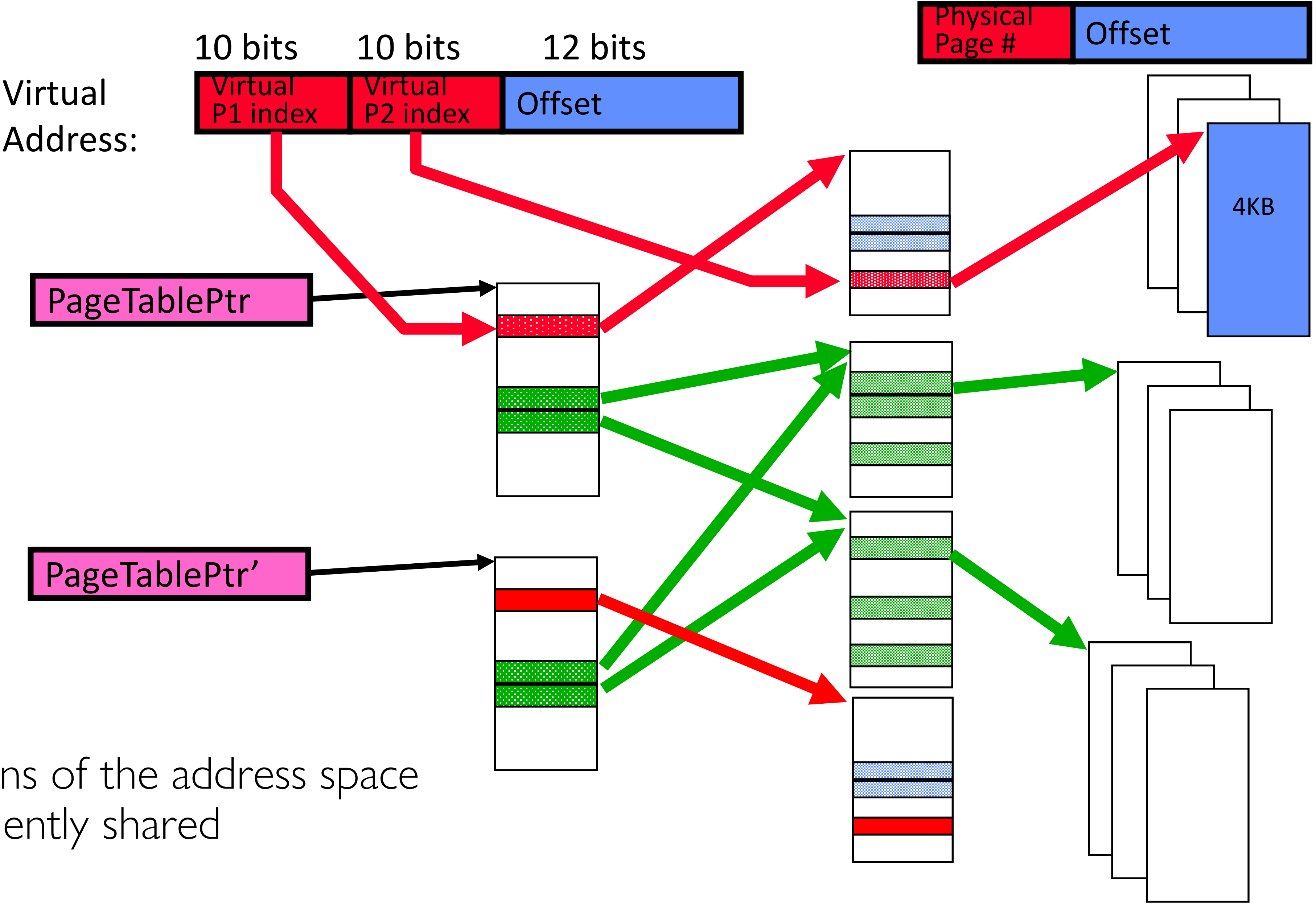
Page Frame Number (Physical Page Number)	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as “valid” bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
PS: Page Size: PS=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

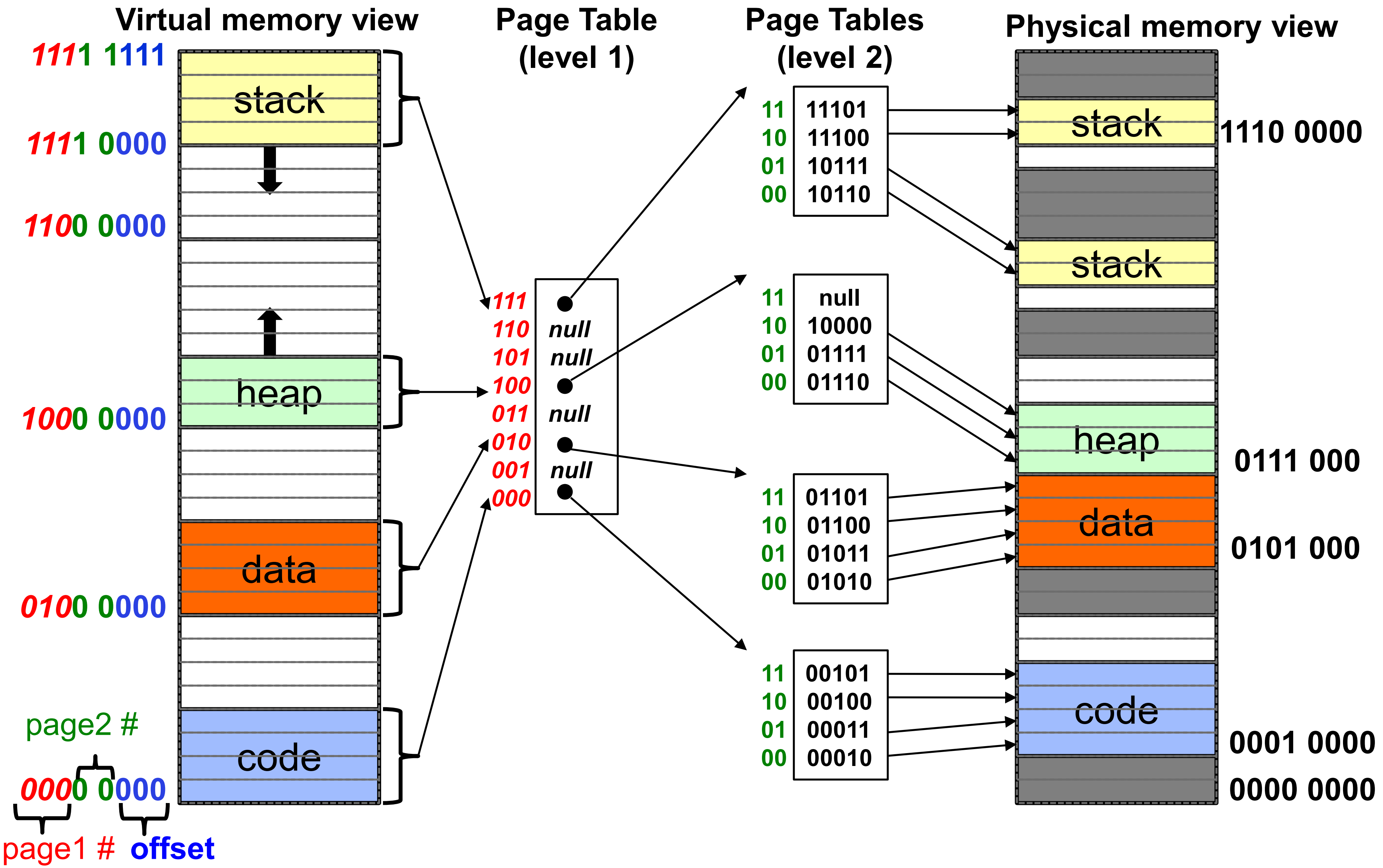
- How do we use the PTE?
  - Invalid PTE can imply different things:
    - Region of address space is actually invalid or
    - Page/directory is just somewhere else than memory
  - Validity checked first
    - OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - Address spaces disconnected after child created
  - How to do this cheaply?
    - Make copy of parent's page tables (point at same memory)
    - Mark entries in both sets of page tables as read-only
    - Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

Sharing with multilevel page tables



- Entire regions of the address space can be efficiently shared

Summary: Two-Level Paging



Page Table (level 1)

111	●
110	null
101	null
100	●
011	null
010	●
001	null
000	●

Page Tables (level 2)

11	11101
10	11100
01	10111
00	10110

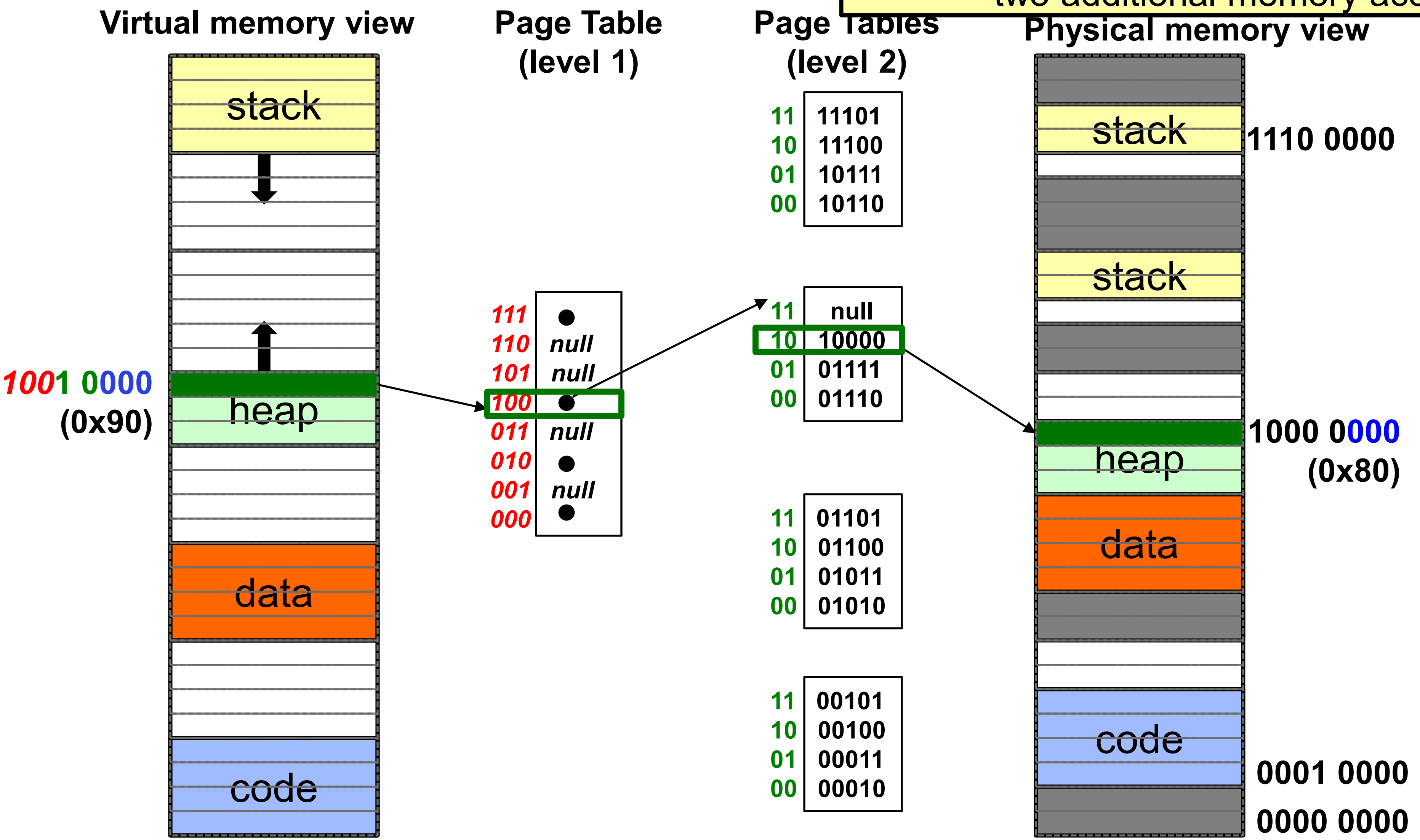
11	null
10	10000
01	01111
00	01110

11	01101
10	01100
01	01011
00	01010

11	00101
10	00100
01	00011
00	00010

Summary: Two-Level Paging

In best case, total size of page tables ≈ number of pages used by program virtual memory. Requires two additional memory accesses!



Page Table (level 1)

111	●
110	null
101	null
100	●
011	null
010	●
001	null
000	●

Page Tables (level 2)

11	11101
10	11100
01	10111
00	10110

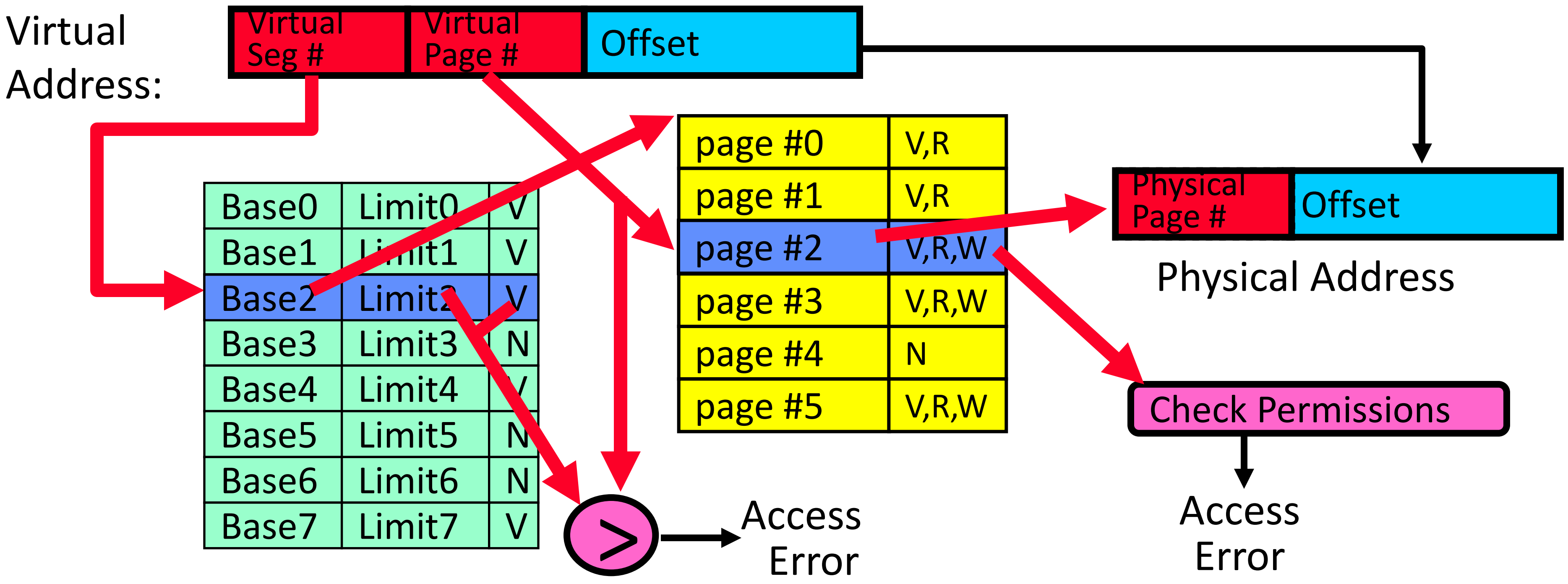
11	null
10	10000
01	01111
00	01110

11	01101
10	01100
01	01011
00	01010

11	00101
10	00100
01	00011
00	00010

Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

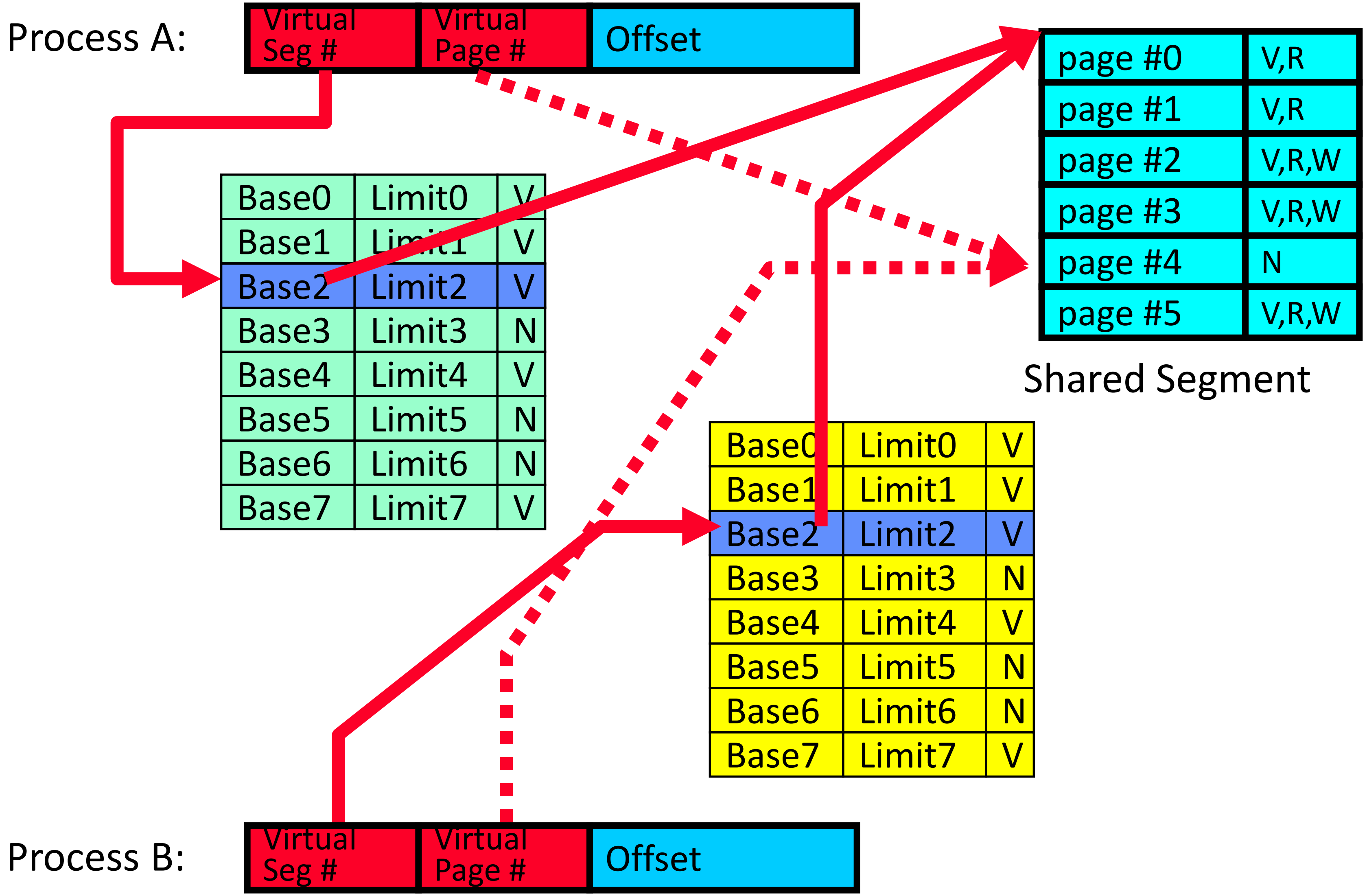


Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

What about Sharing (Complete Segment)?



Process A:

Virtual Seg #	Virtual Page #	Offset

Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

Shared Segment

Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

Process B:

Virtual Seg #	Virtual Page #	Offset

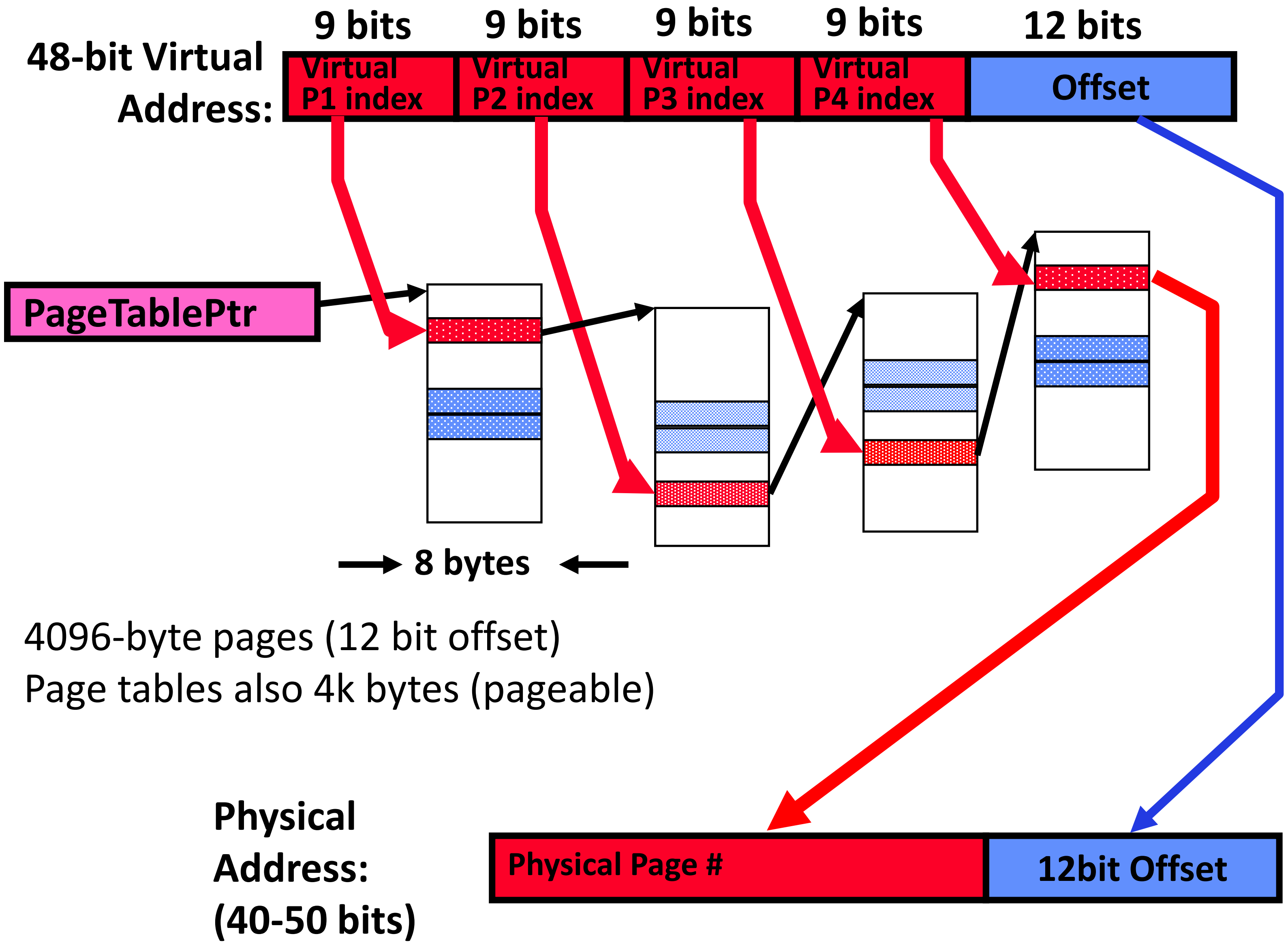
Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - However, the 10b-10b-12b configuration keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - Seems very expensive!

Recall: Dual-Mode Operation

- Can a process modify its own translation tables? NO!
  - If it could, could get access to all of physical memory (no protection!)
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bit(s) in control register only accessible in Kernel mode
  - Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode (more in moment)
- Note that x86 model actually has more modes:
  - Traditionally, four "rings" representing priority; most OSes use only two:
    - » Ring 0 ⇒ Kernel mode, Ring 3 ⇒ User mode
    - » Called "Current Privilege Level" or CPL
  - Newer processors have additional mode for hypervisor ("Ring -1")
- Certain operations restricted to Kernel mode:
  - Modifying page table base (CR3 in x86), and segment descriptor tables
    - » Have to transition into Kernel mode before you can change them!
  - Also, all page-table pages must be mapped only in kernel mode

X86_64: Four-level page table!

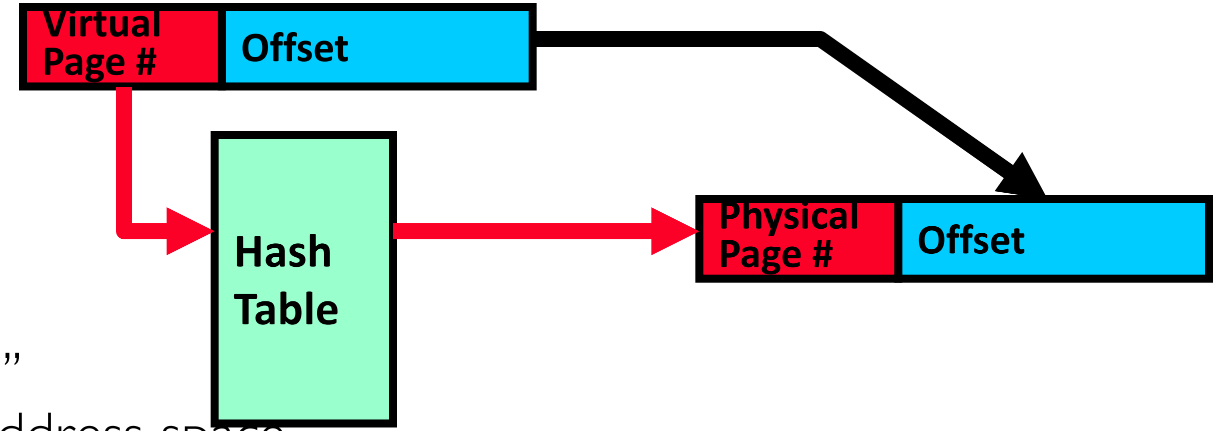


4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

Alternative: Inverted Page Table

Total size of page table ≈ number of pages used by program in physical memory. Hash more complex

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
    - » PowerPC, UltraSPARC, IA64
- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table

Address Translation Comparison

	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Multi-Level Paging		
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table