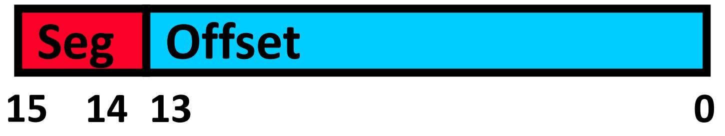
CSE150
Operating Systems
Lecture 14

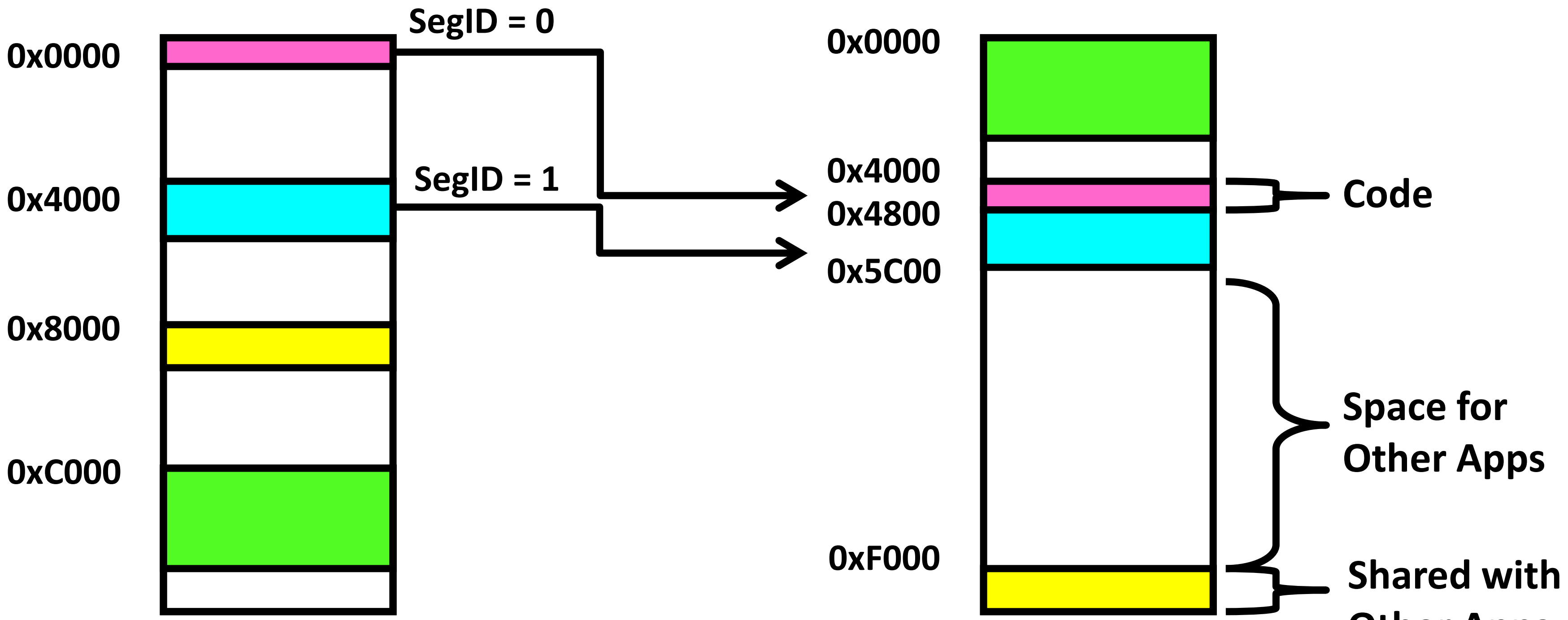
Address Translation (cont.)

Example: Four Segments (16 bit addresses)



Virtual Address Format

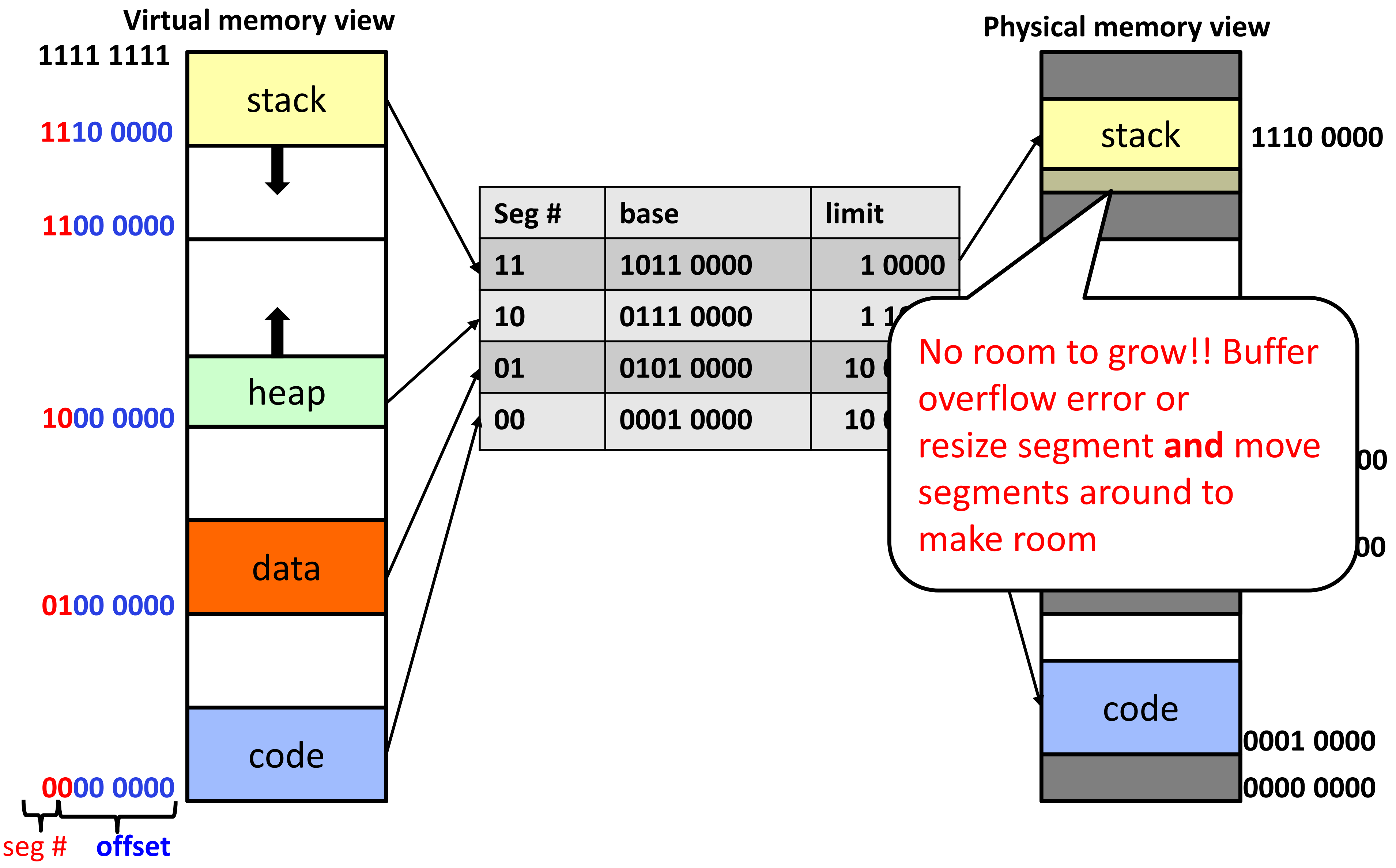
Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Virtual
Address Space

Physical
Address Space

Address Segmentation

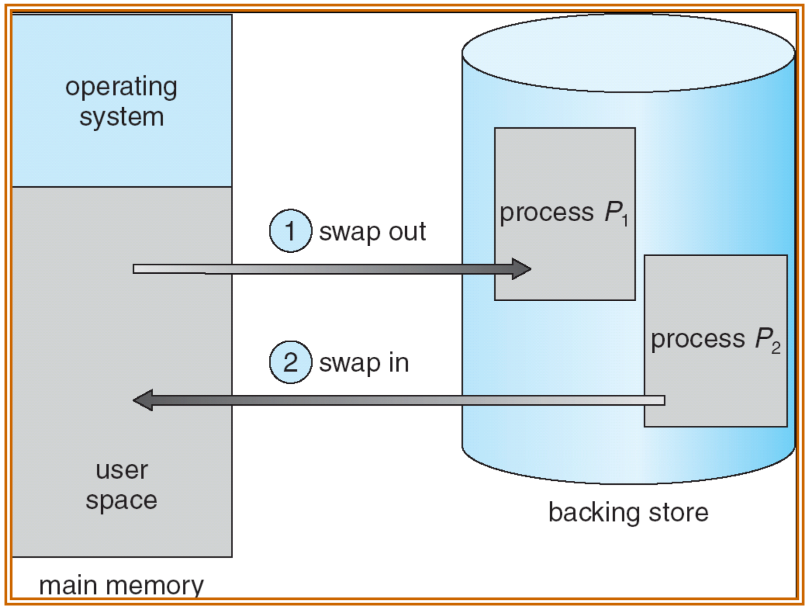


Seg #	base	limit
11	1011 0000	1 0000
10	0111 0000	1 1
01	0101 0000	10
00	0001 0000	10

No room to grow!! Buffer overflow error or resize segment **and** move segments around to make room

Schematic View of Swapping

- Q: What if not all processes fit in memory?
- A: Swapping: Extreme form of Context Switch
  - In order to make room for next process, some or all of the previous process is moved to disk
  - This greatly increases the cost of context-switching



- 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

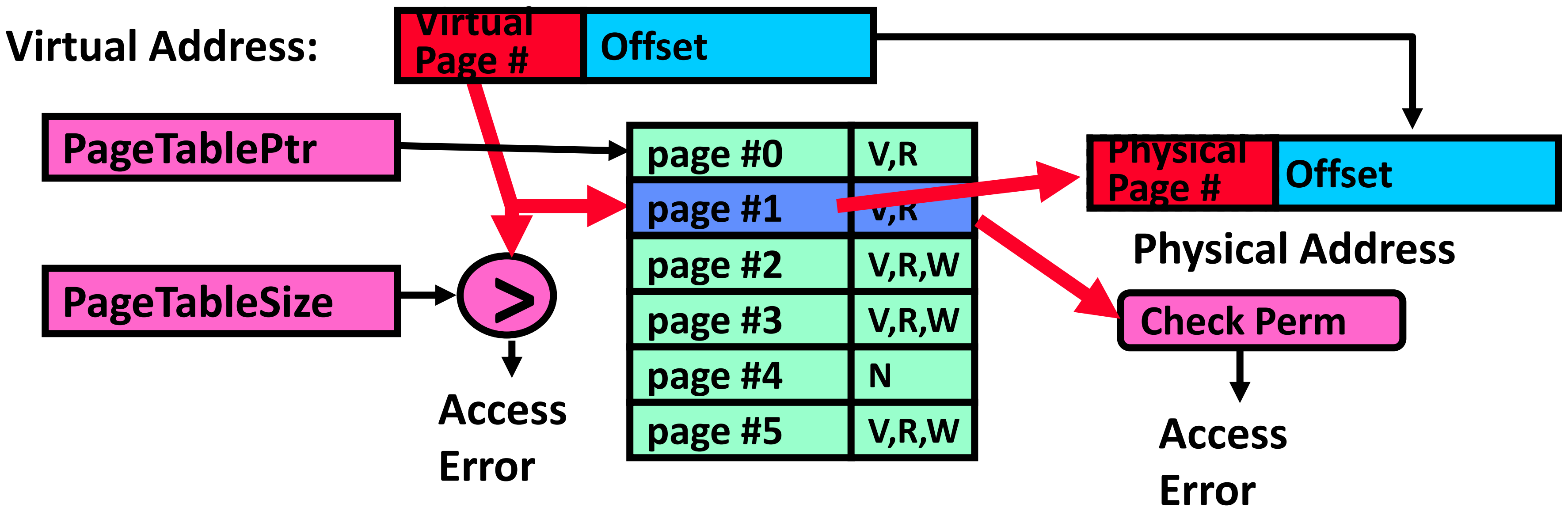
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don’t need all memory within allocated chunks

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

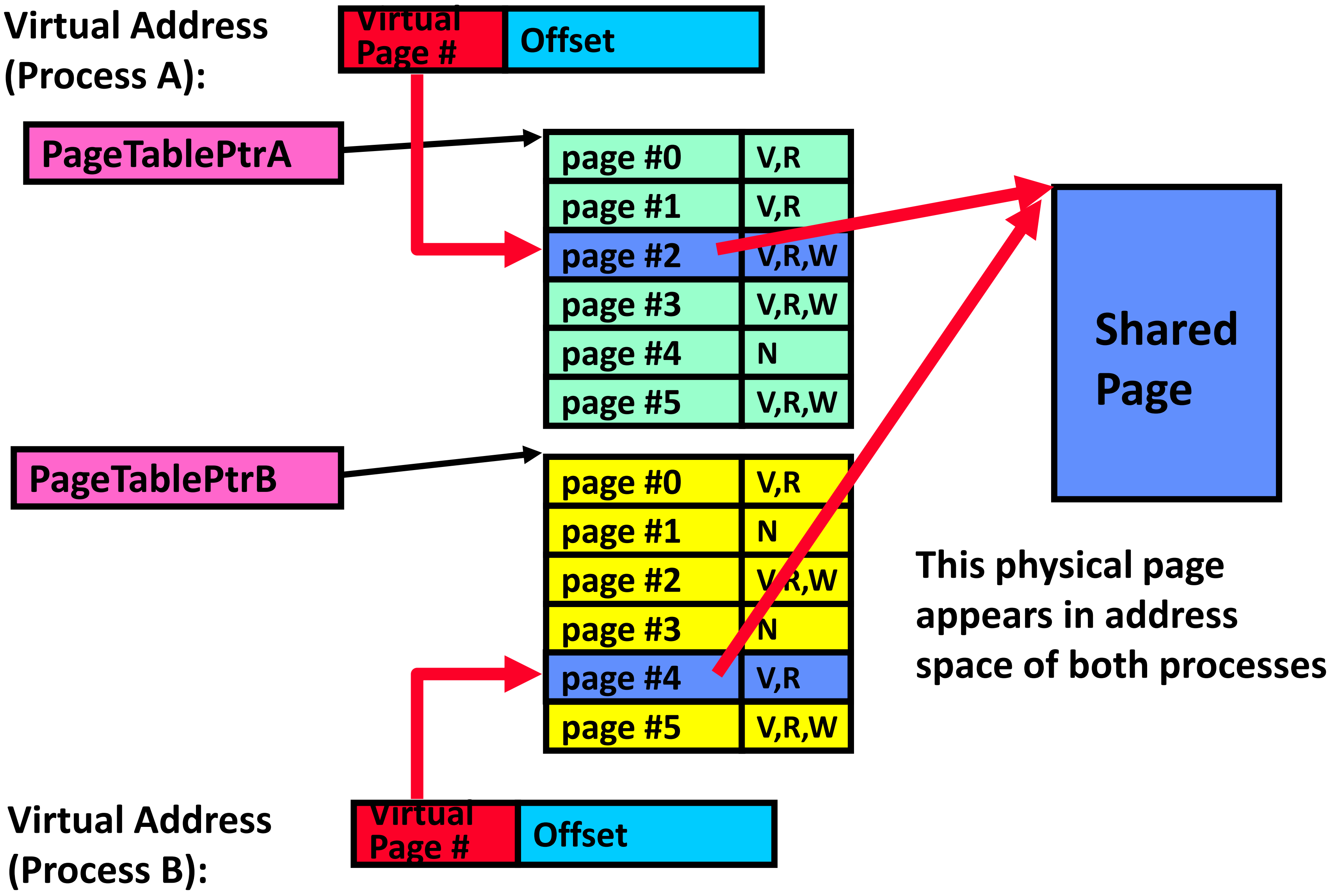
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?



- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset $\Rightarrow$ 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

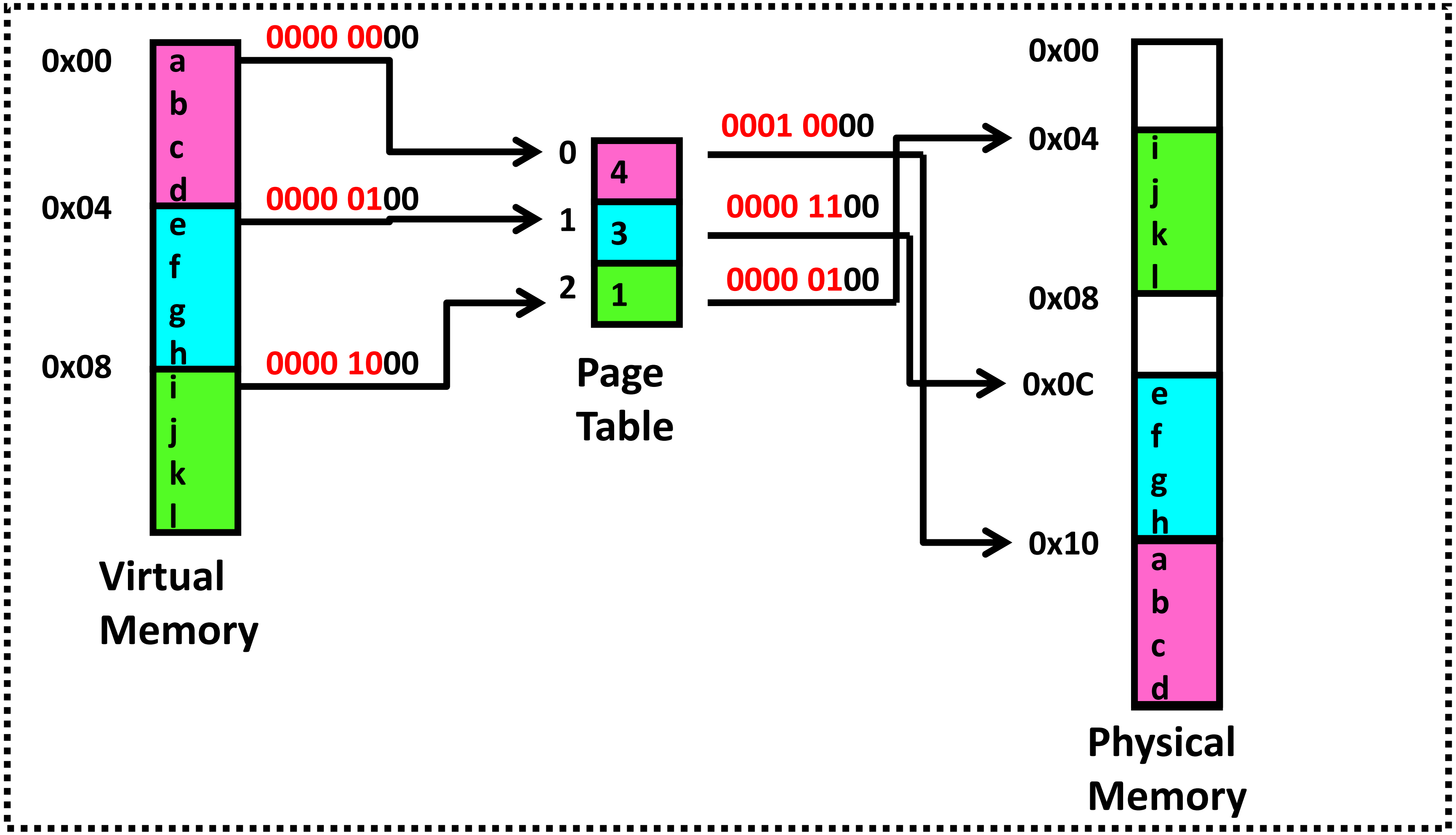
What about Sharing?



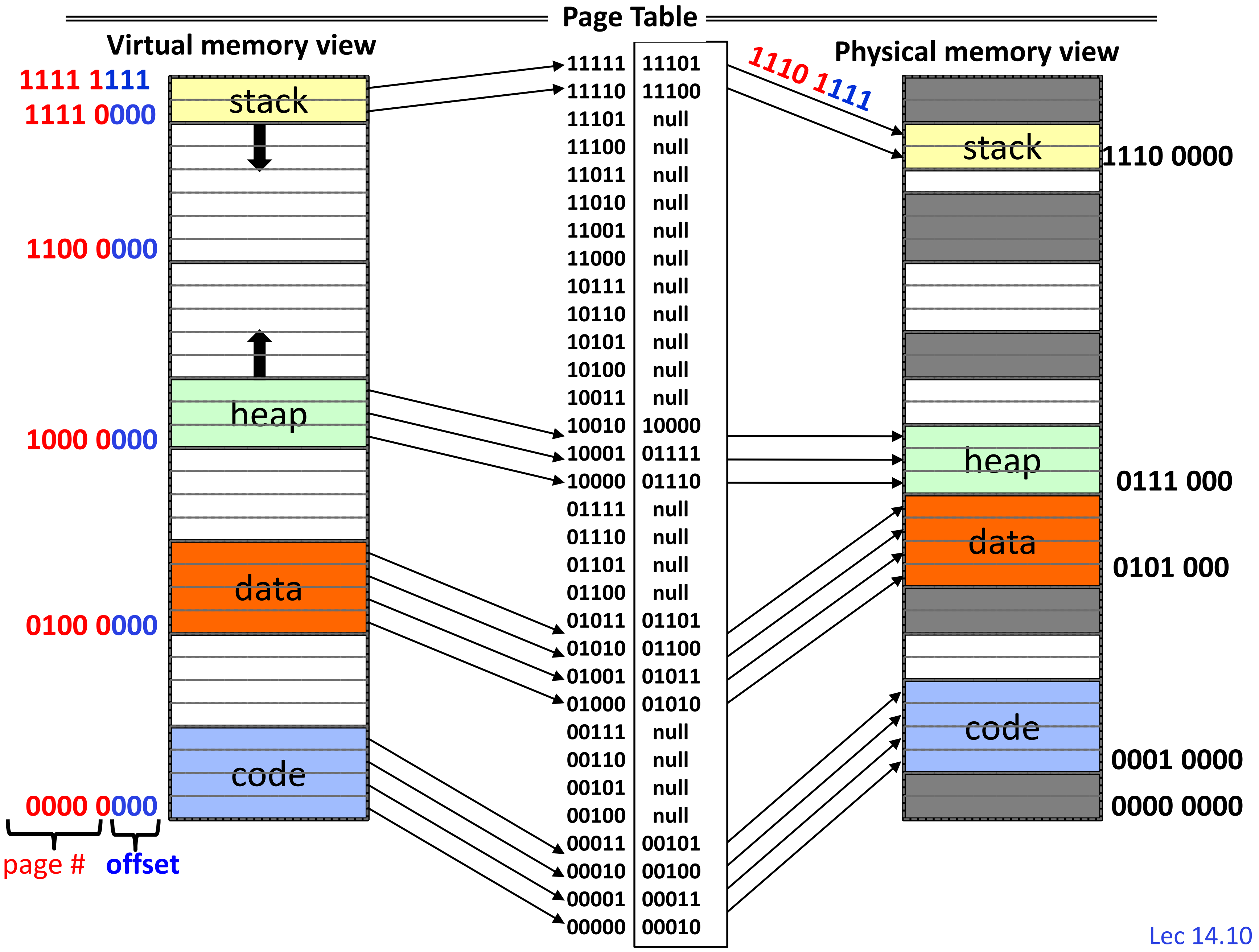
This physical page appears in address space of both processes

Simple Page Table Example

Example (4 byte pages, 8-bit memory)

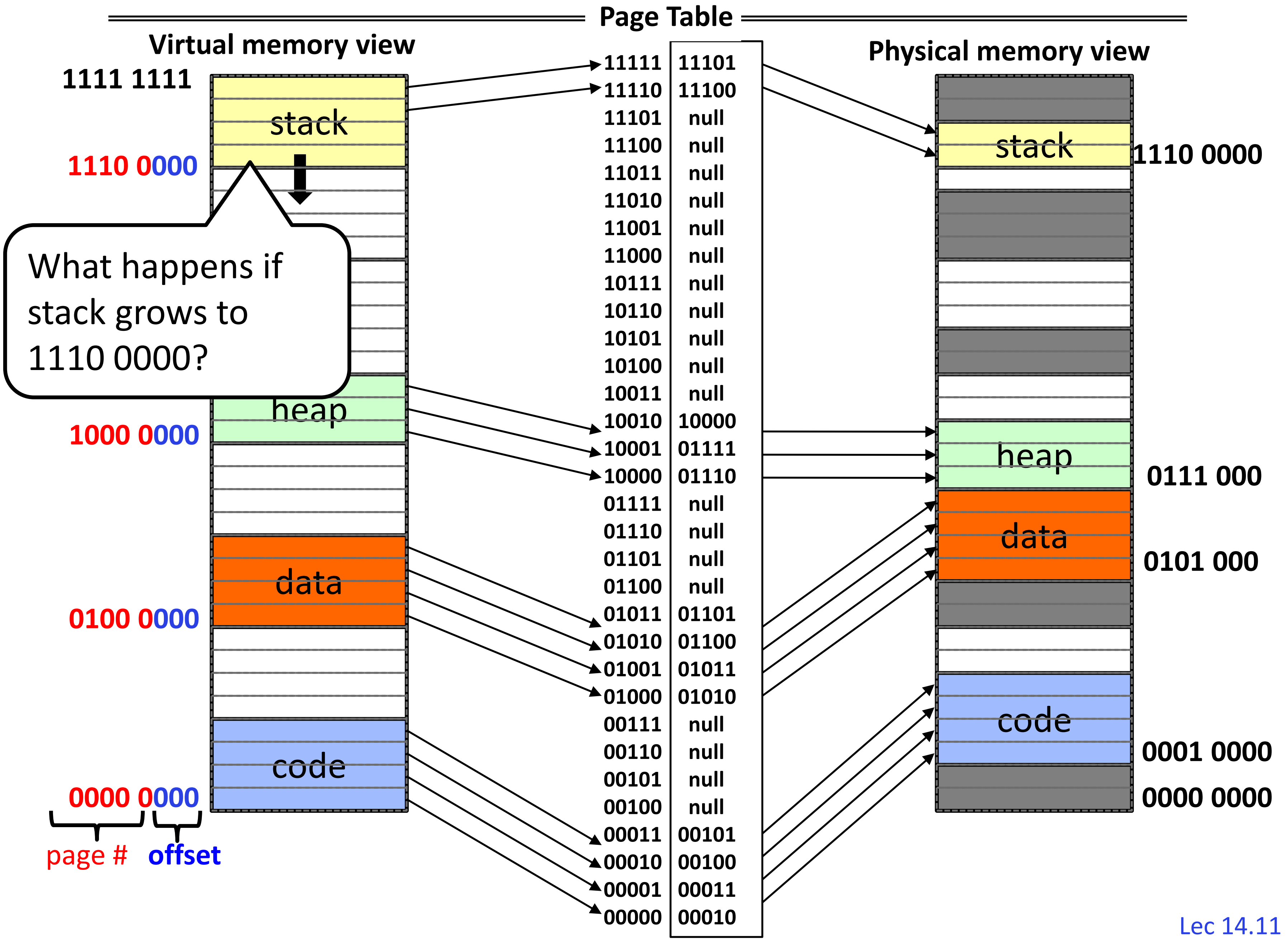


Paging



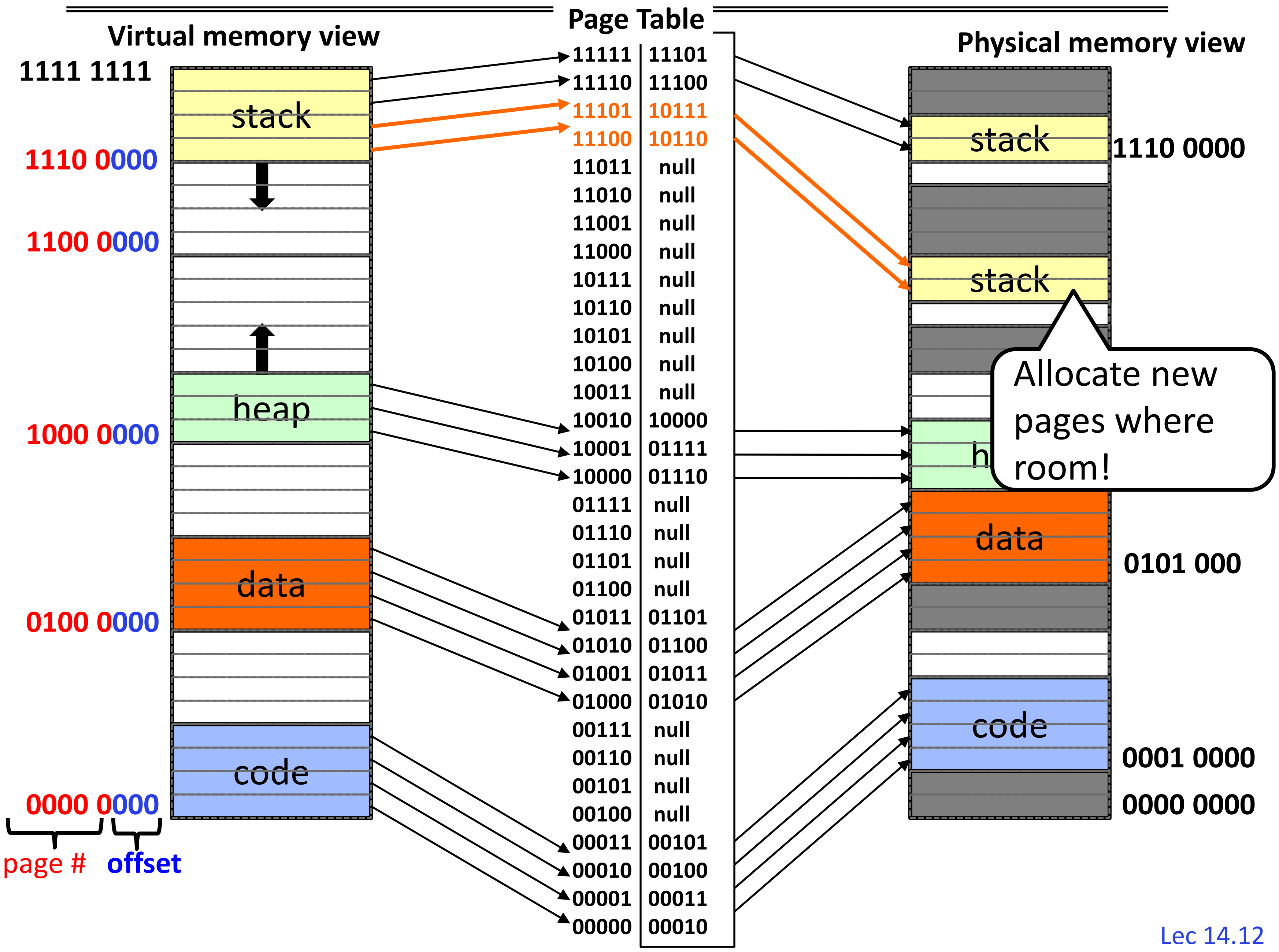
11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101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

Paging



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

Paging



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

Page Table Discussion

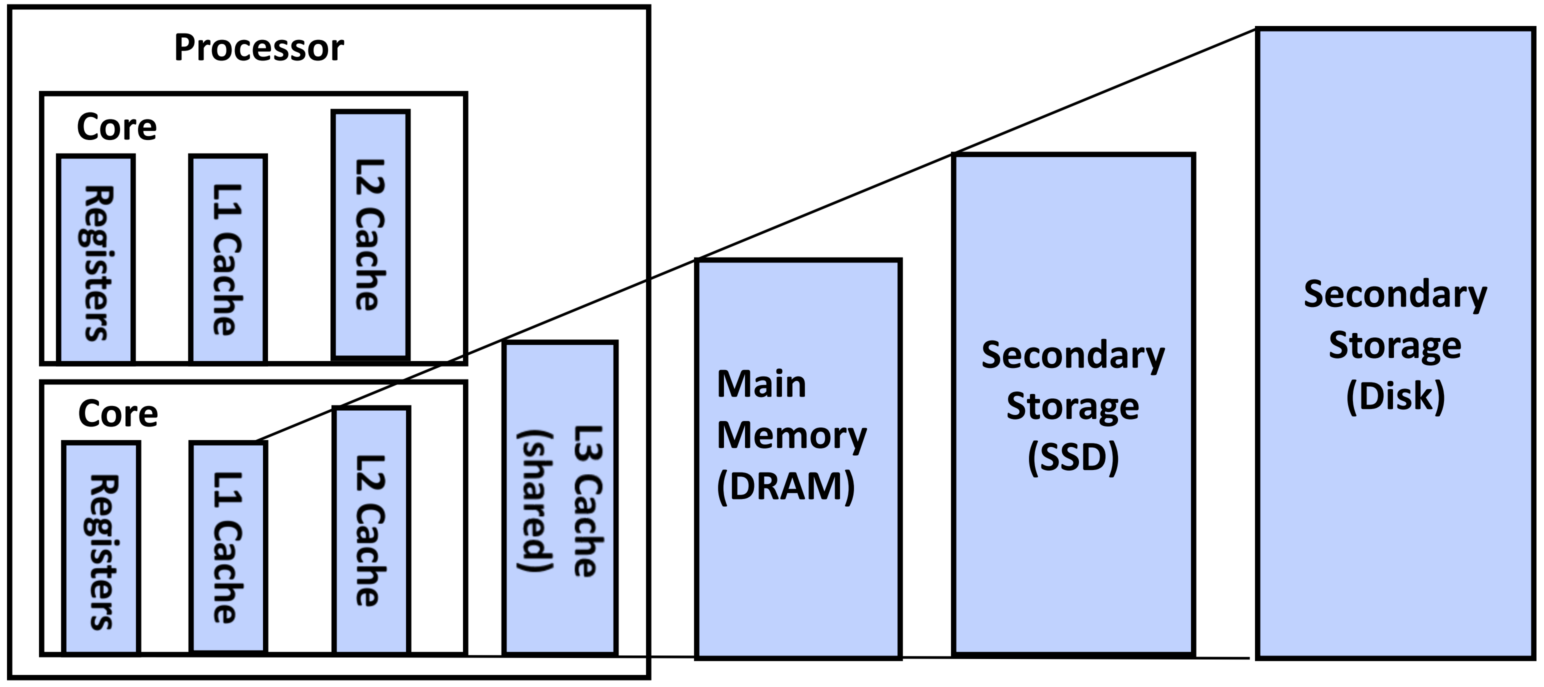
- What needs to be switched on a context switch?
  - Page table pointer

- 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 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time $\Rightarrow$ would be nice to have working set of page table in memory

- How about combining paging and segmentation?

Memory Hierarchy

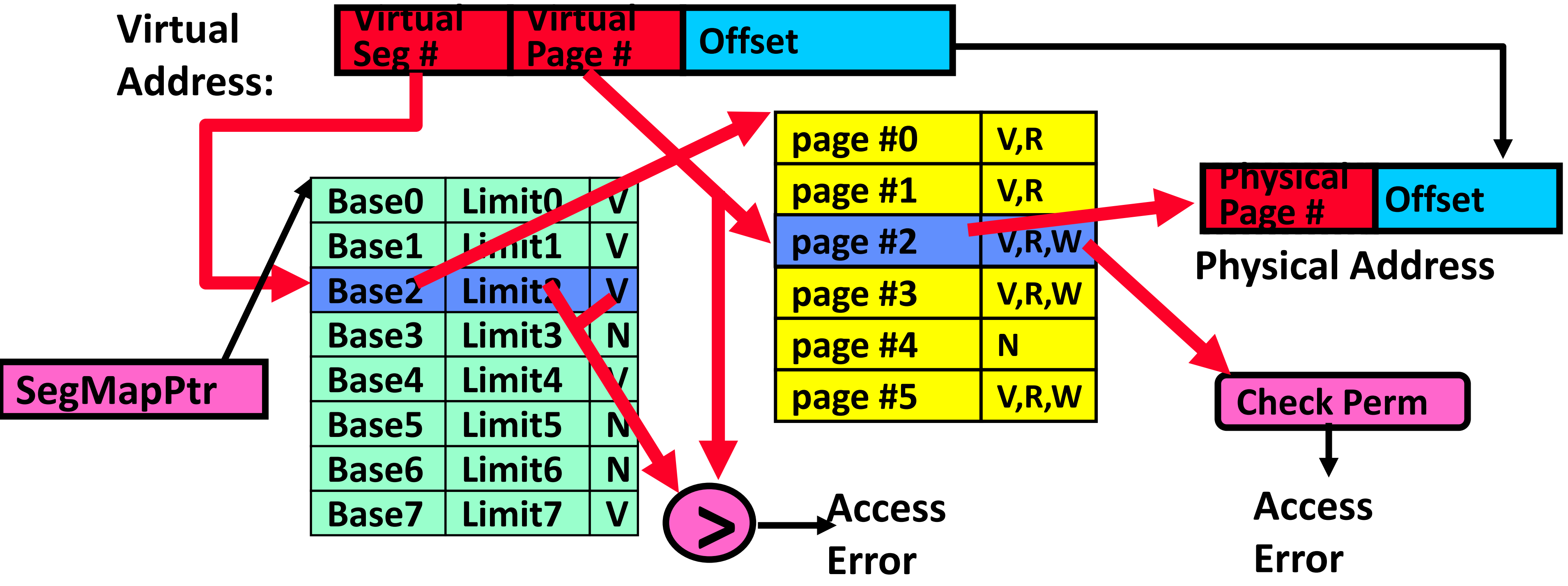
- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

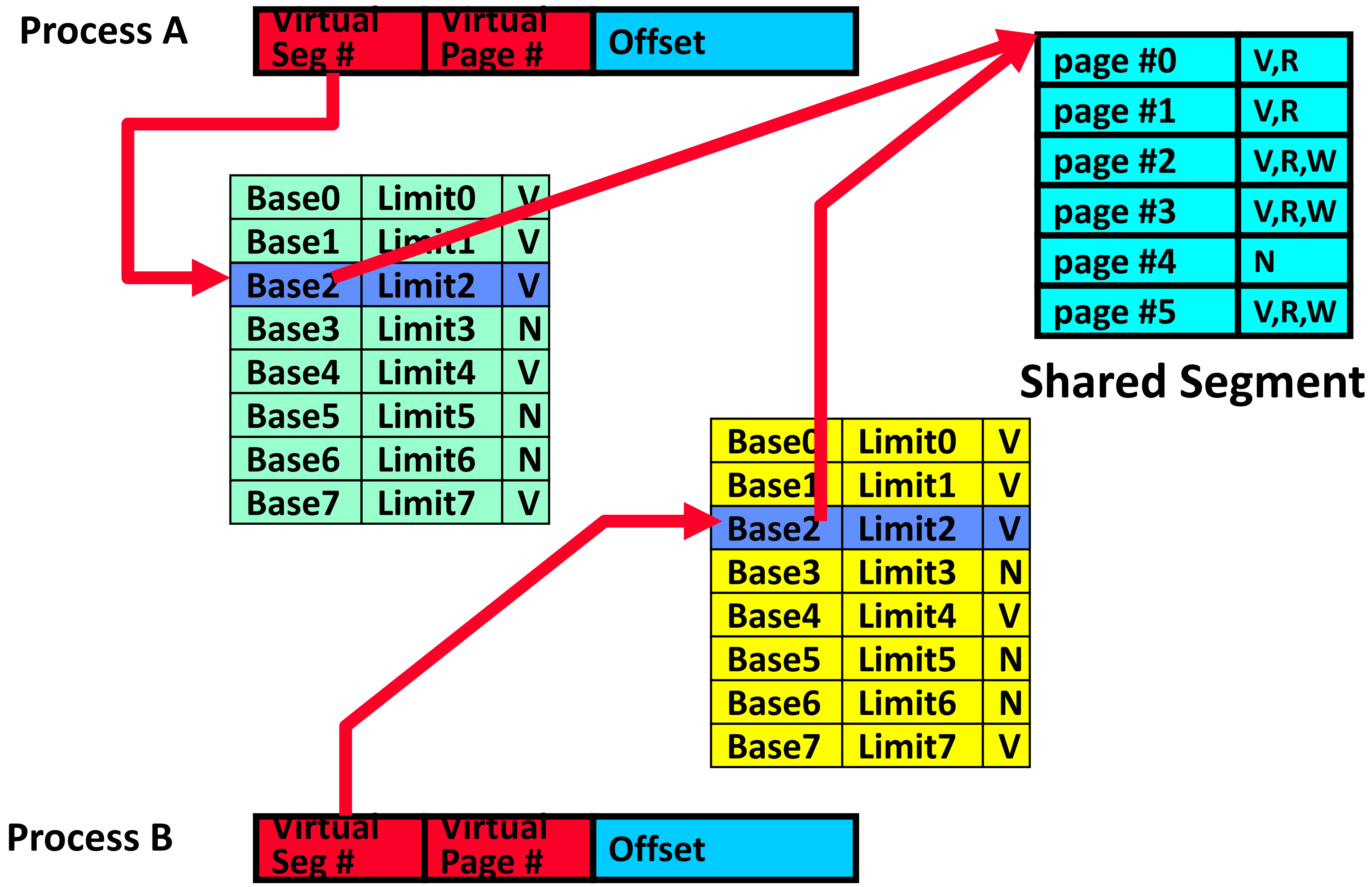
Multi-level Translation

- 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):



- What must be saved/restored on context switch?
  - Segment map pointer register (for this example)

What about Sharing (Complete Segment)?

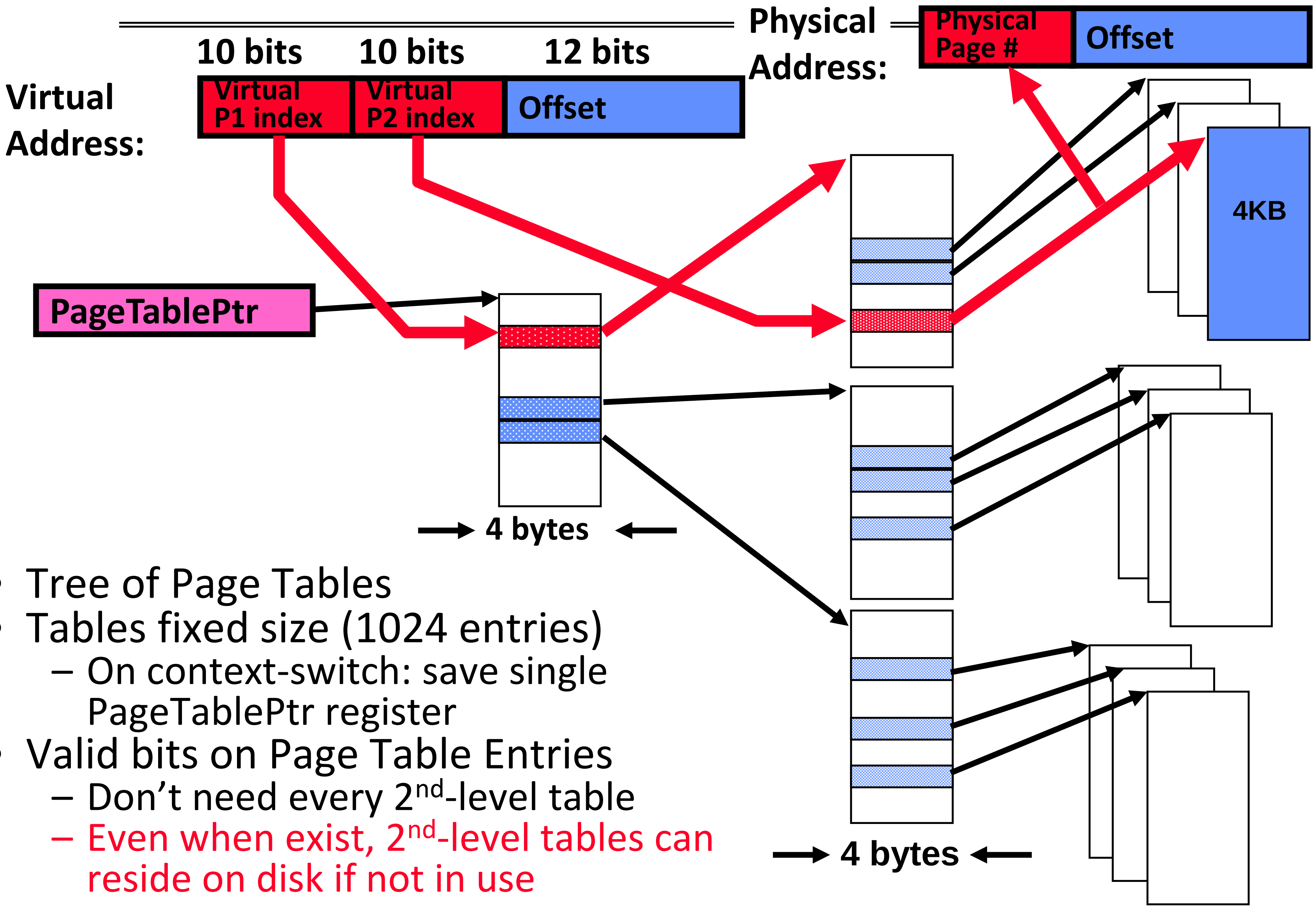


Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

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

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

Another common example: two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- 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

What is in a 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”

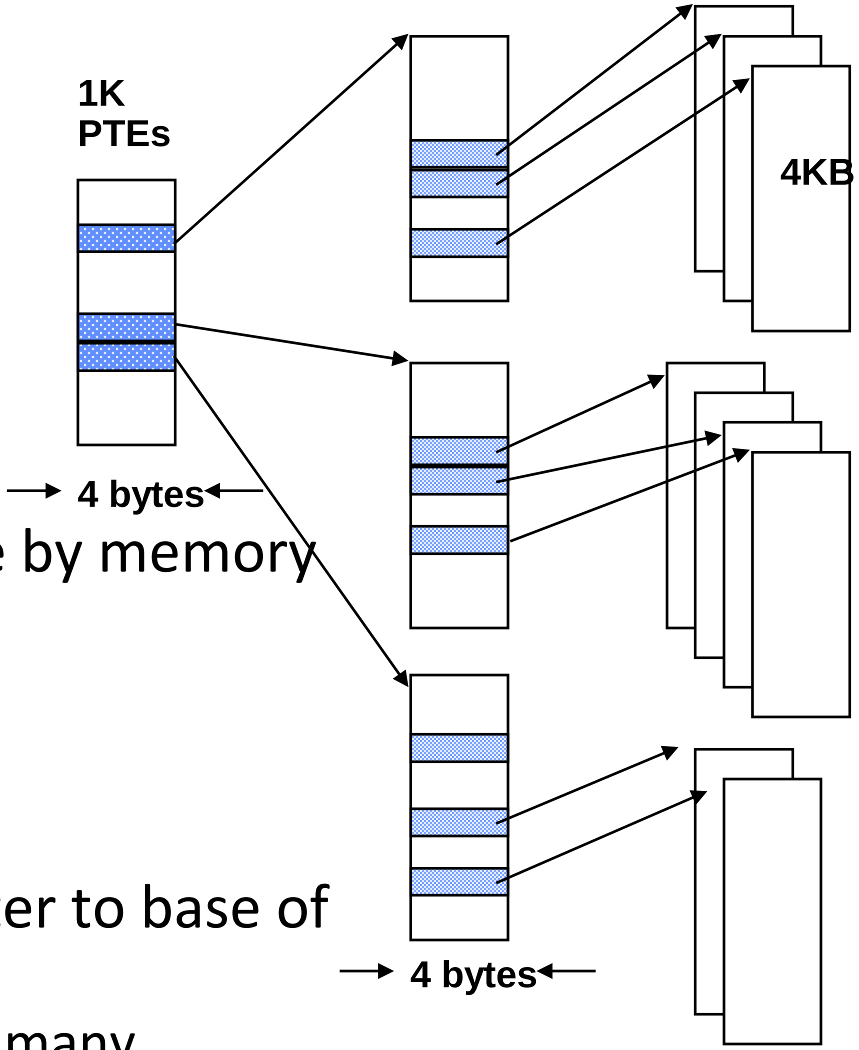
Page Frame Number (Physical Page Number)	**Free (OS)**	**0**	**L**	**D**	**A**	**PCD**	**PWT**	**U**	**W**	**P**
31-12	**11-9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

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
L:	L=1⇒4MB page (directory only). Bottom 22 bits of virtual address serve as offset

Example of Translation Table Format

Two-level Page Tables
32-bit address:

10	10	12
P1 index	**P2 index**	**page offset**

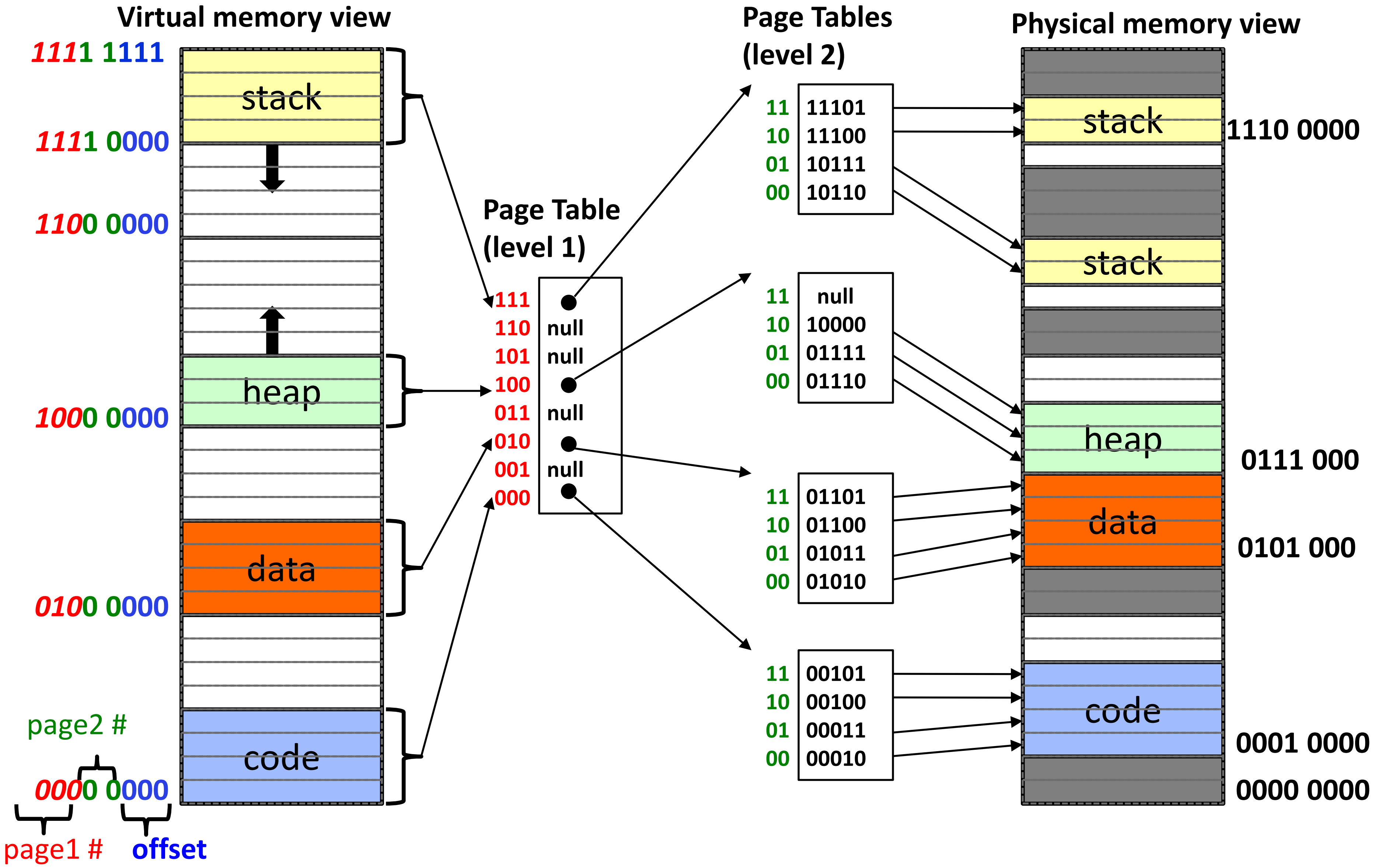


- Page: a unit of memory translatable by memory management unit (MMU)
  - Typically 1K – 8K
- Page table structure in memory
  - Each user has different page table
- Address Space switch: change pointer to base of table (hardware register)
  - Hardware traverses page table (for many architectures)
  - MIPS uses software to traverse table

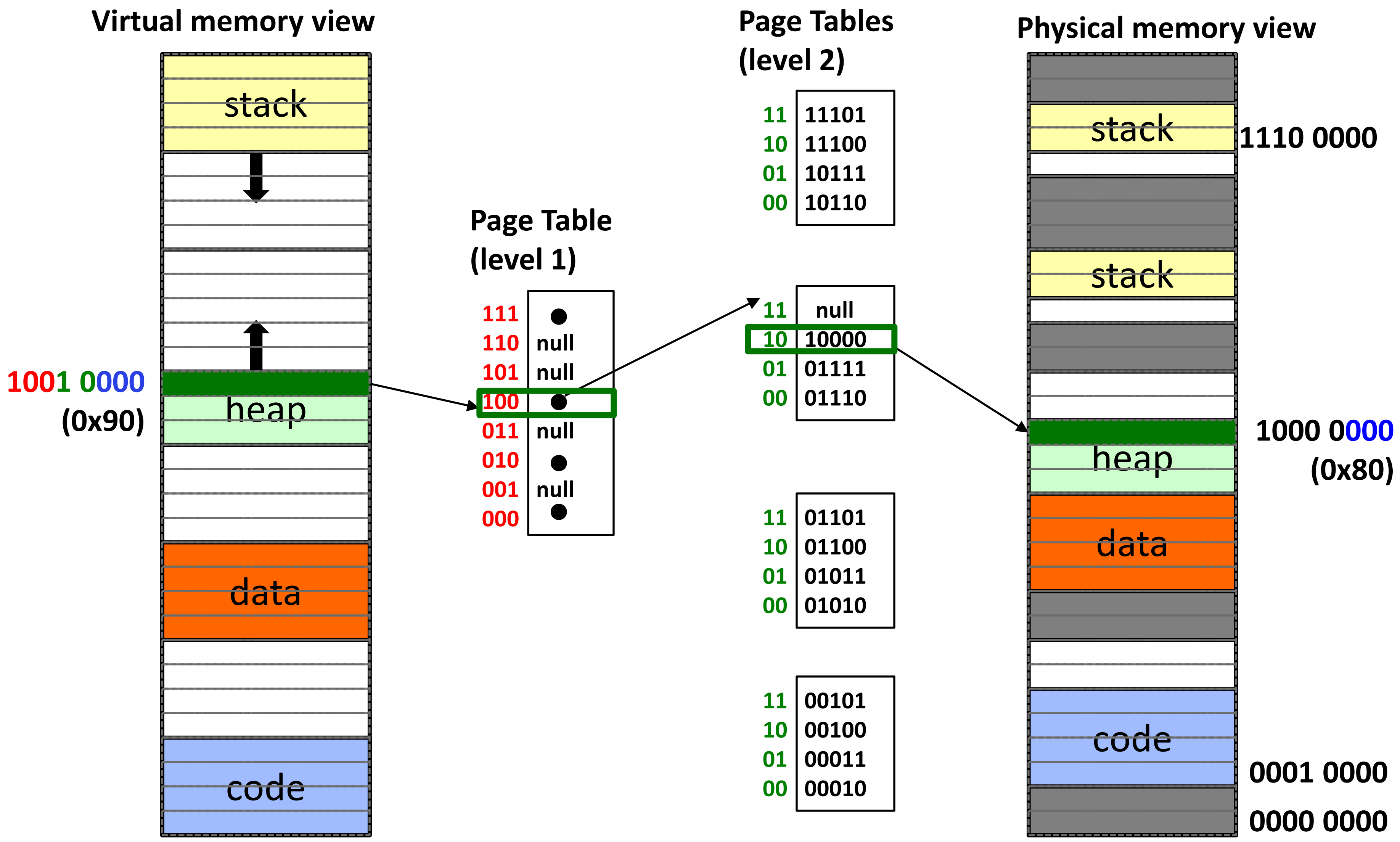
Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application – size is proportional to usage
    - » In other words, 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, previous example keeps tables to exactly one page in size
  - Three (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

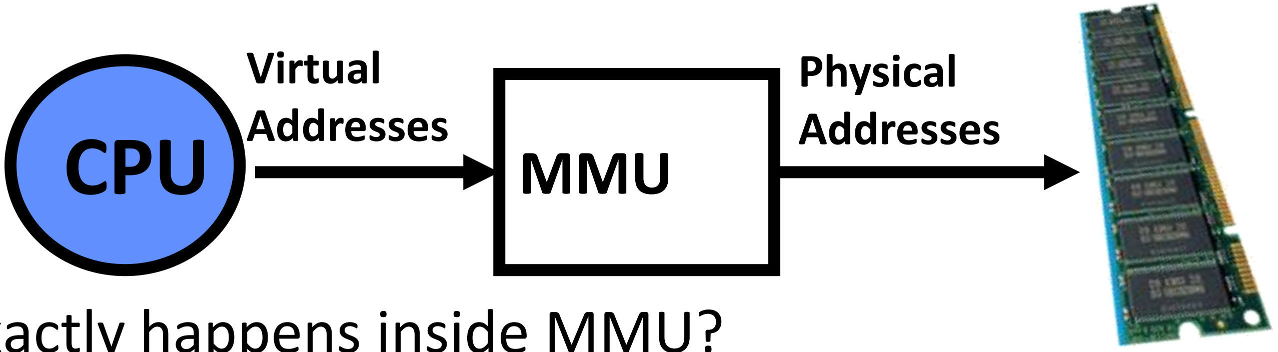
Summary: Two-Level Paging



Summary: Two-Level Paging



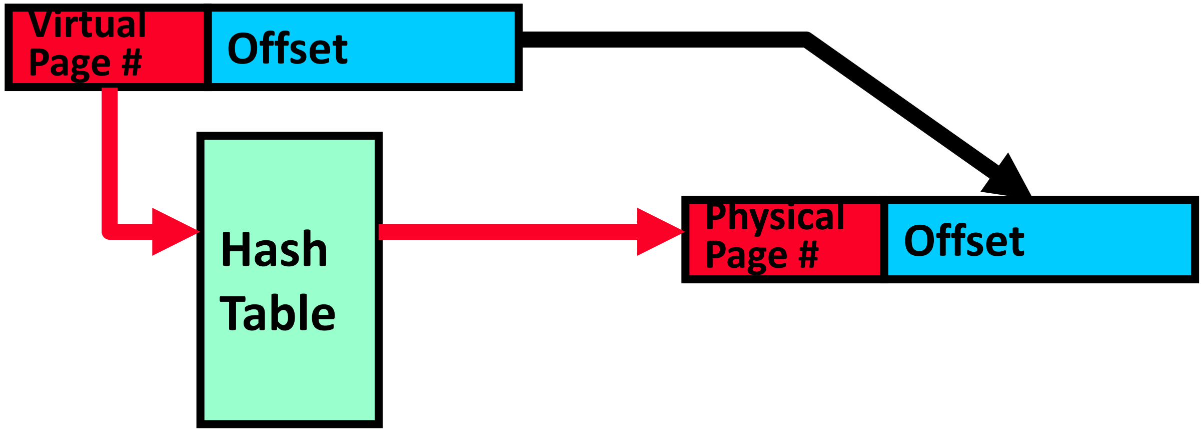
How is the translation accomplished?



- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
  - For each virtual address, takes page table base pointer and traverses the page table in hardware
  - Generates a "Page Fault" if it encounters invalid PTE
    - » Fault handler will decide what to do
    - » More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Expensive!
- In fact, need way to cache translations for either case!

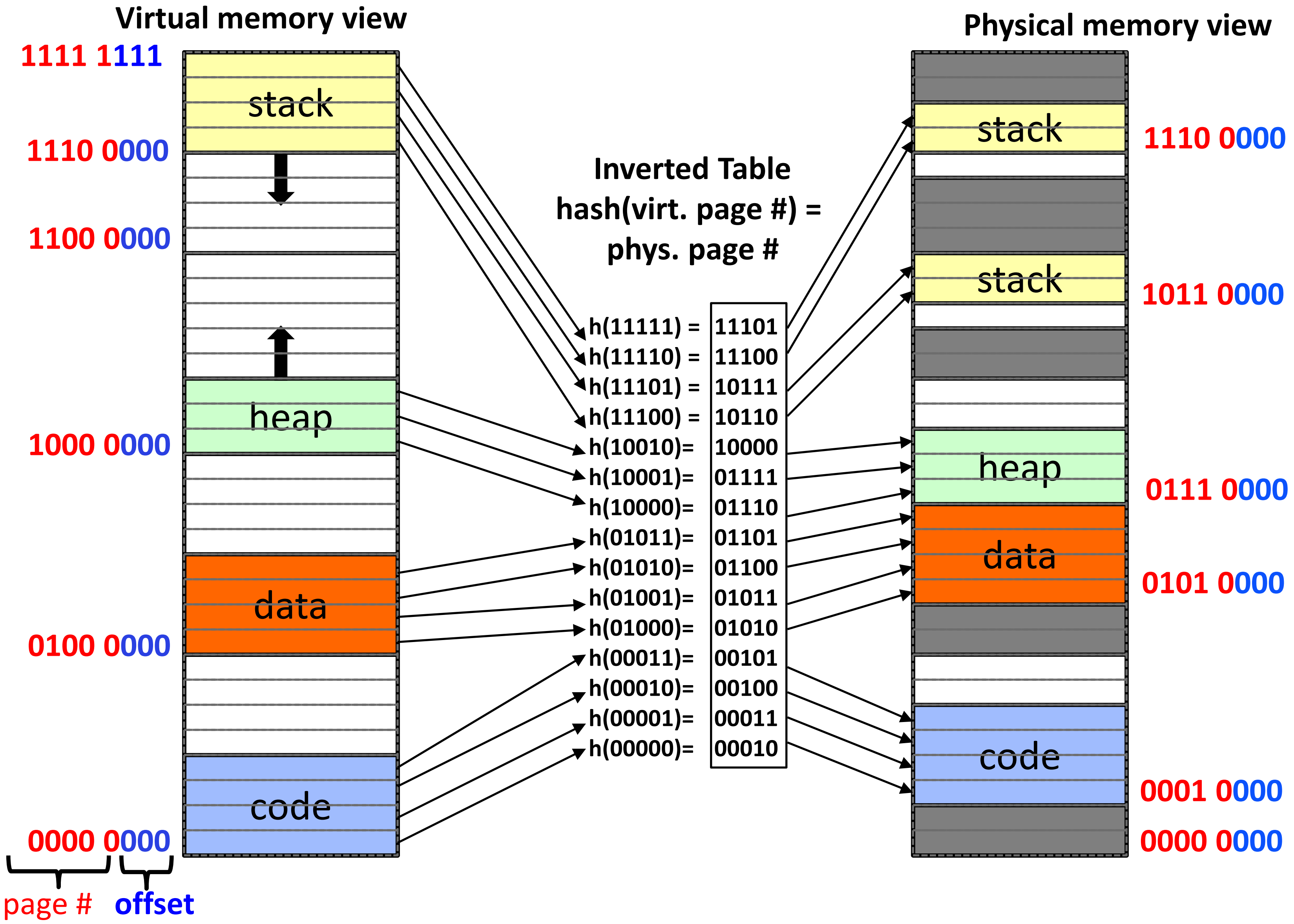
Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is almost 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
  - Instead, size directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces (IA64)
- Cons: Complexity of managing hash changes
  - Often in hardware!

Summary: Inverted Table



Summary

	Advantages	**Disadvantages**
Segment	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory
Paged segmentation	Table size ~ # of pages in virtual memory, fast easy allocation	Multiple memory references per page access
Two-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex