Segments, Pages

11 January 2026
Lecture 11

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Deadlock

Starvation

Deadlock Graph

Deadlock conditions

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlock detection algorithm

Bankers Algorithm

Topics for Today

- Address Spaces and Segmentation
  - Fragmentation
- Page tables

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

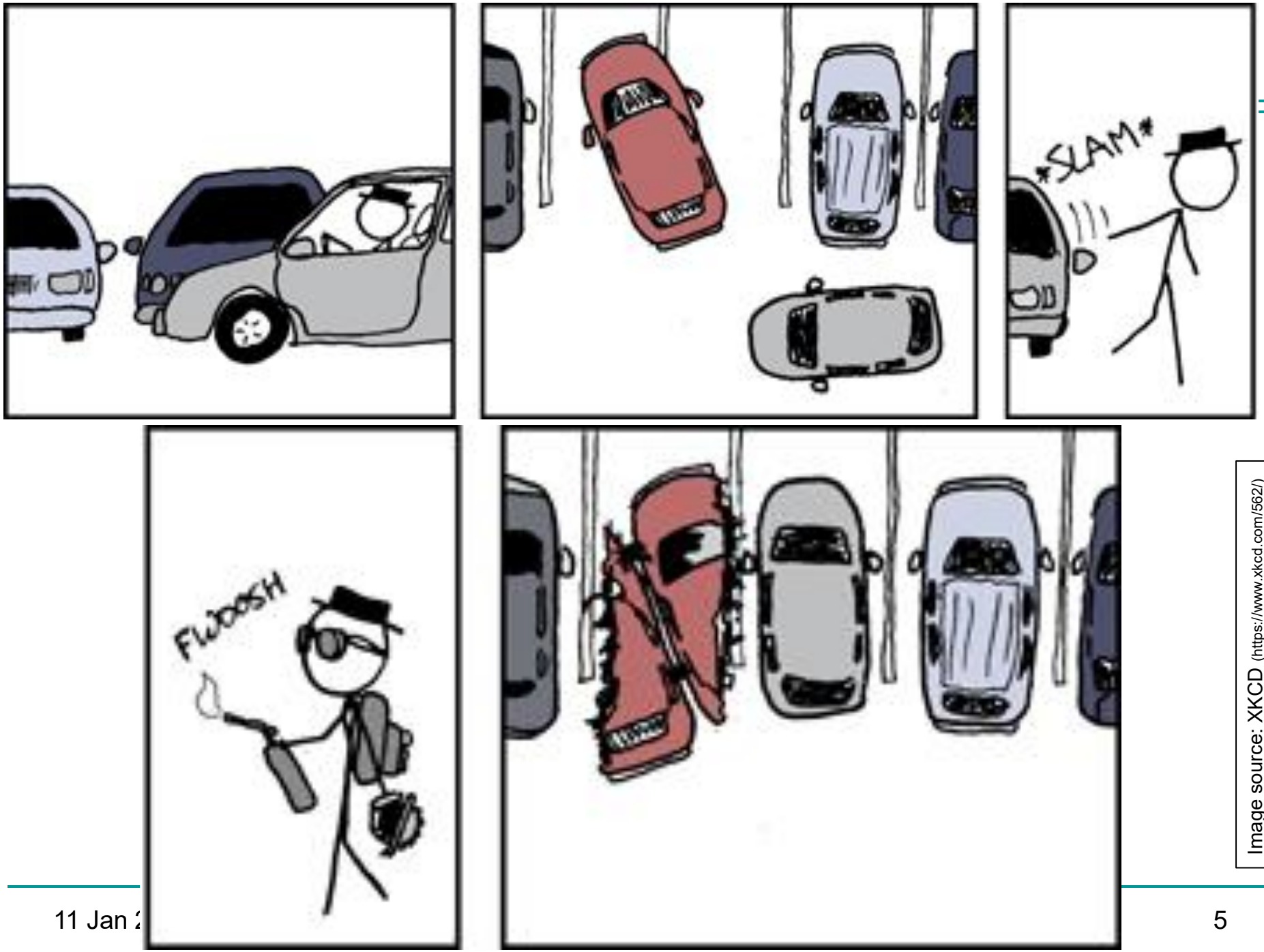


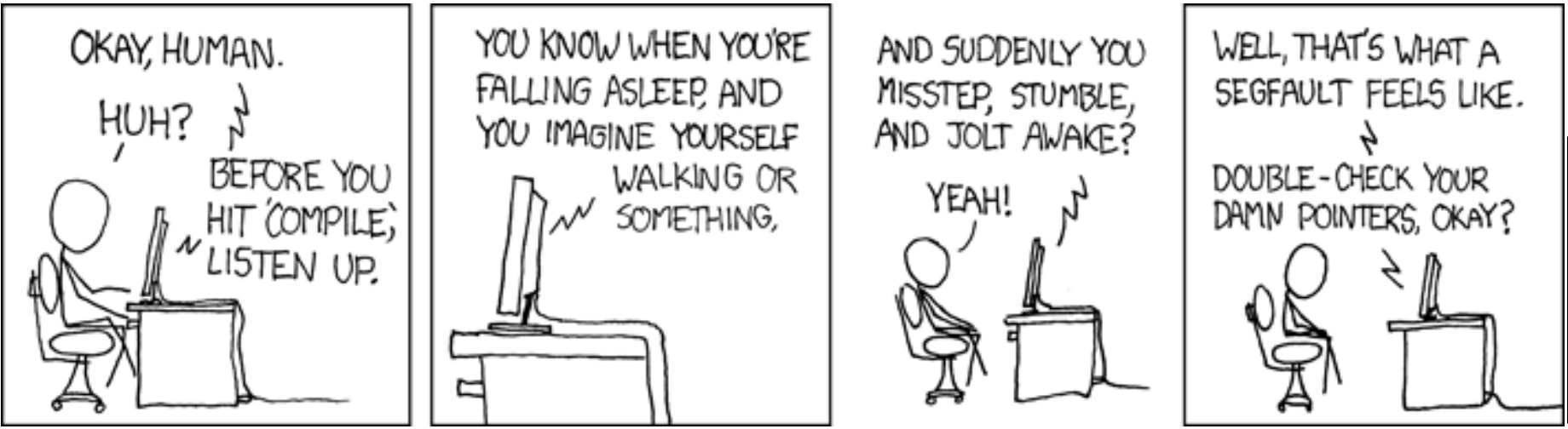
Image source: XKCD (https://www.xkcd.com/562/)

Image source: http://imgur.com/OFEy7

Segmentation Fault

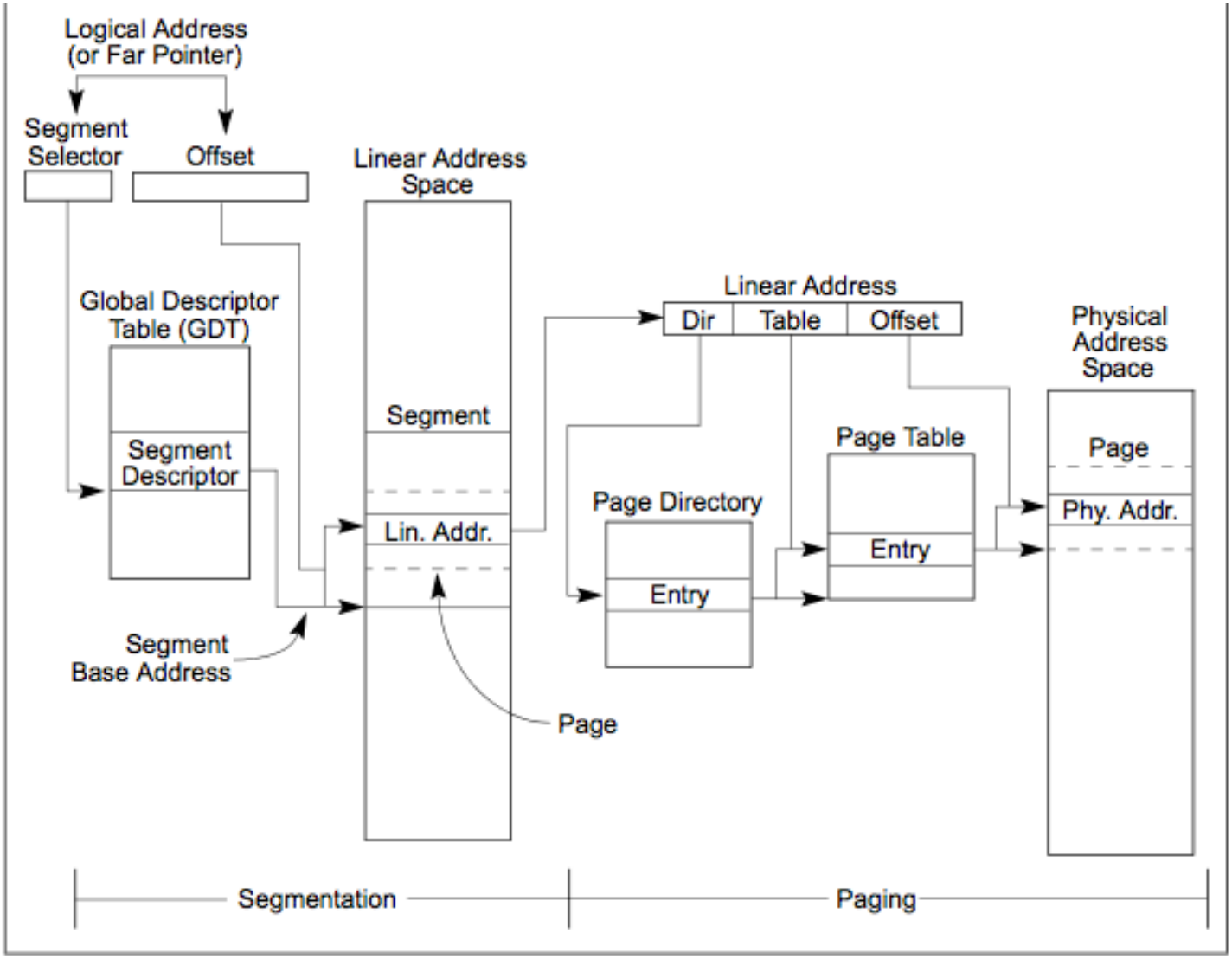
Image source: http://www.computerhope.com/jargon/s/segfault.png

```
user@host:/tmp/segfault$ cat segfault.c
void main() {
  char *str = "Hello, world!";
  *str = 'A';
}
user@host:/tmp/segfault$ gcc segfault.c -o segfault
user@host:/tmp/segfault$ ./segfault
Segmentation fault
neil@snap:/tmp/segfault$
```



Source: XKCD

Making it real:
x86 Memory model with segmentation (16/32-bit)



X86 Segment Descriptors (32-bit Protected Mode)

- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
  - There are 6 registers: SS, CS, DS, ES, FS, GS
- What is in a segment register?
  - A *pointer* to the actual segment description:

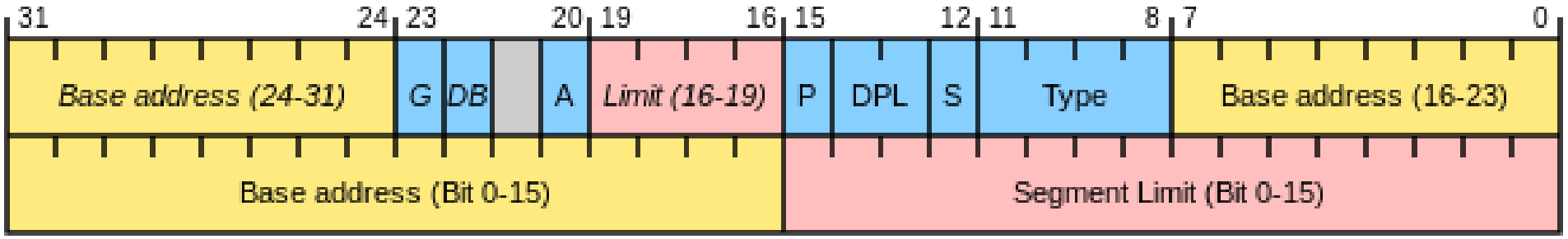
Segment Selector [13 bits]	G/L	RPL

G/L selects between GDT and LDT tables (global vs local descriptor tables)

- Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
  - Includes length of table (for < 2^{13}) entries

X86 Segment Descriptors (32-bit Protected Mode)

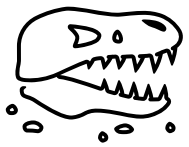
- Descriptor format (64 bits):



G: Granularity of segment (0: 16bit, 1: 4KiB unit)
DB: Default operand size (0; 16bit, 1: 32bit)
A: Freely available for use by software
P: Segment present
DPL: Descriptor Privilege Level
S: System Segment (0: System, 1: code or data)
Type: Code, Data, Segment

How are segments used?

- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process



In legacy applications (16-bit mode):

- Segments provide protection for different components of user programs
- Separate segments for chunks of code, data, stacks
- Limited to 64K segments

How are segments used?

- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process

Modern use in 32-bit Mode:

- Segments "flattened", i.e. every segment is 4GB in size
- One exception: Use of GS (or FS) as a pointer to "Thread Local Storage" (TLS)
  - A thread can make accesses to TLS like this:

```
mov eax, gs(0x0)
```

How are segments used?

- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process

Modern use in 64-bit ("long") mode

- Most segments (SS, CS, DS, ES) have zero base and no length limits
- Only FS and GS retain their functionality (for use in TLS)

So Far

- Address Spaces and Segmentation
  - Fragmentation
- Page tables

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:
      `0011000111000l101 … 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?

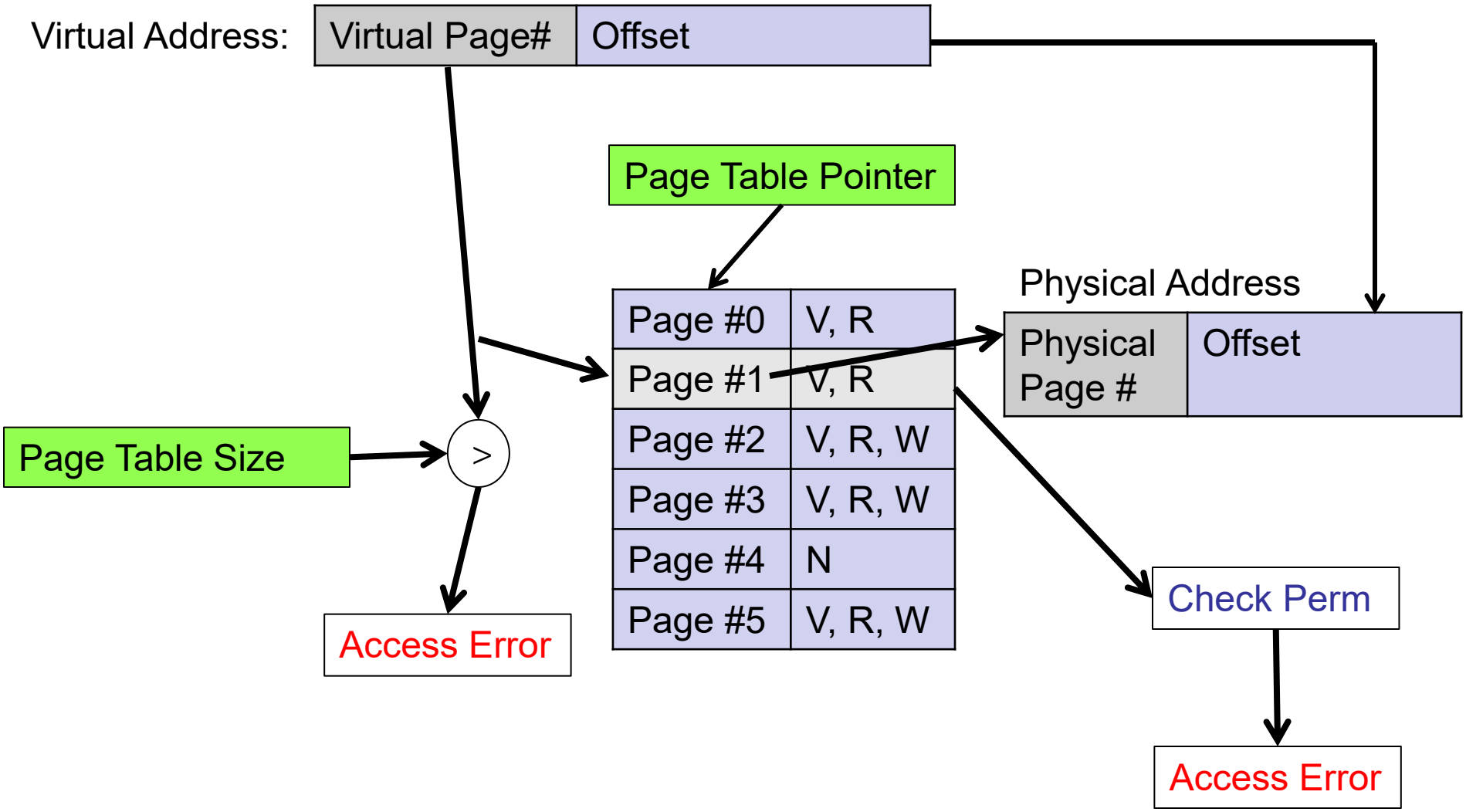
Page Table (One per process)

- Resides in physical memory
- Contains physical page and permission for each virtual page
- Permissions include: Valid bits, Read, Write, etc

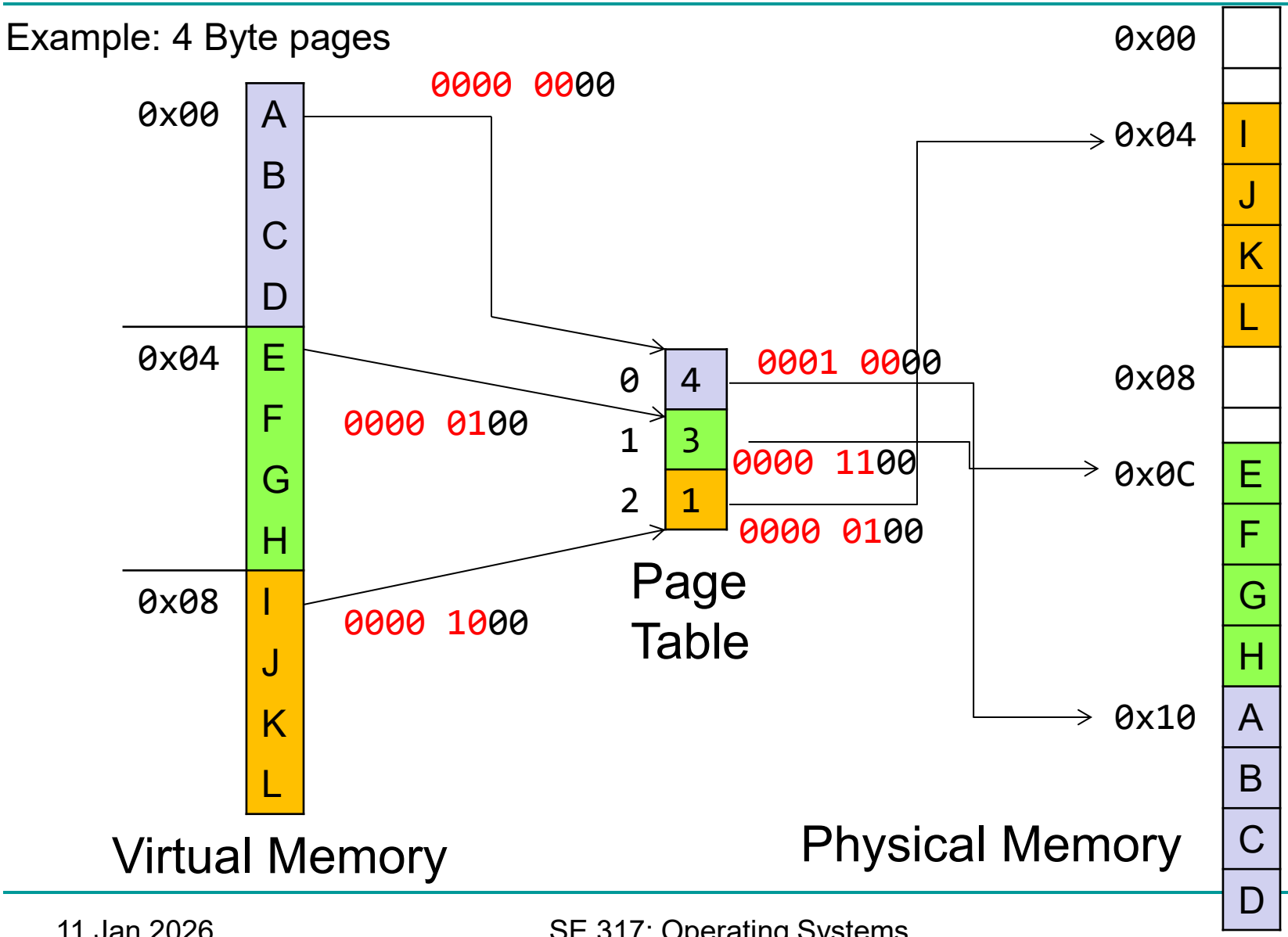
Virtual address mapping

- Offset from Virtual address copied to Physical Address
  - Example: `10 bit` offset ⇒ `1024 byte` pages
- Virtual page # is all remaining bits
  - Example for `32-bits: 32-10 = 22 bits`, (4 million entries)
  - Physical page # copied from table into physical address
- Check Page Table bounds and permissions

Implementing Paging

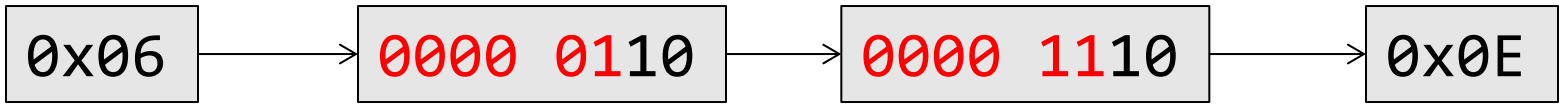


Simple Page Table Example

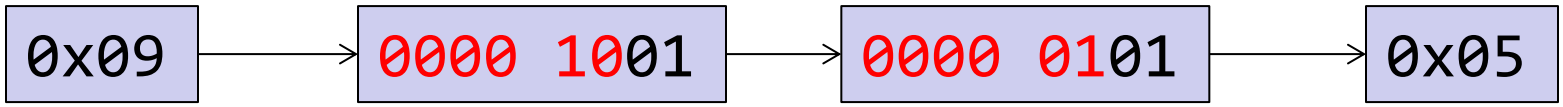


Simple Page Table Example

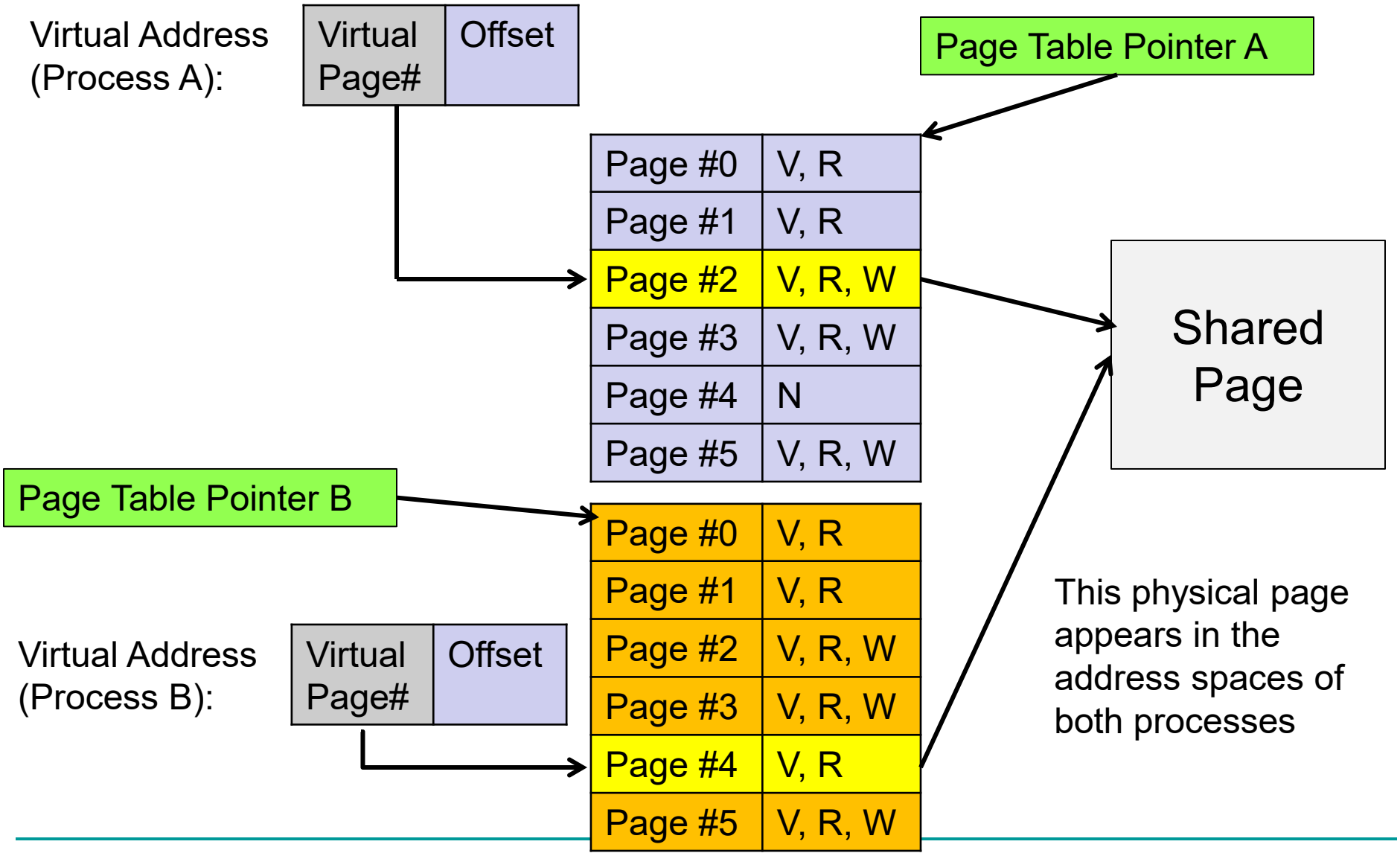
Virtual Address 1

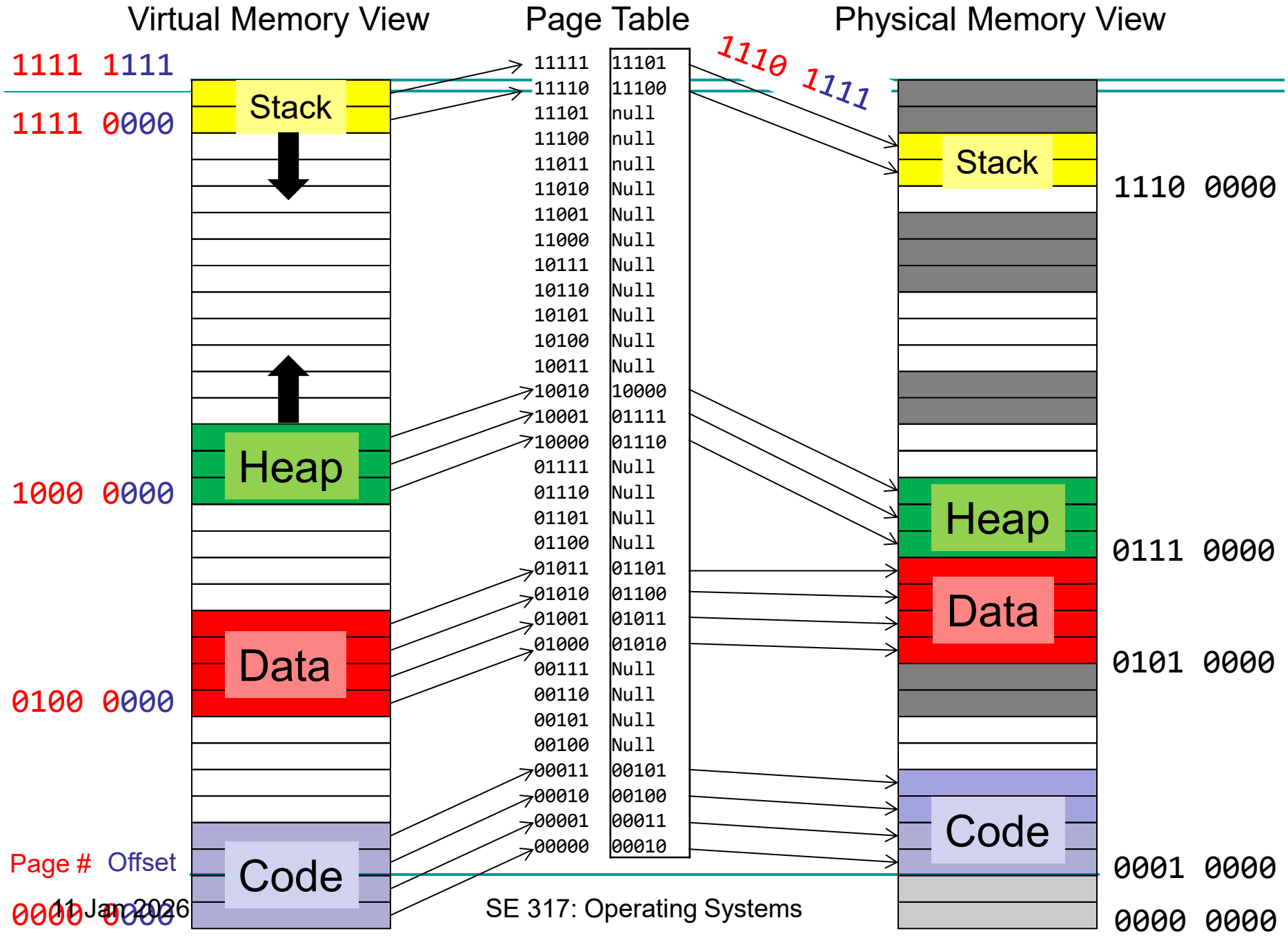


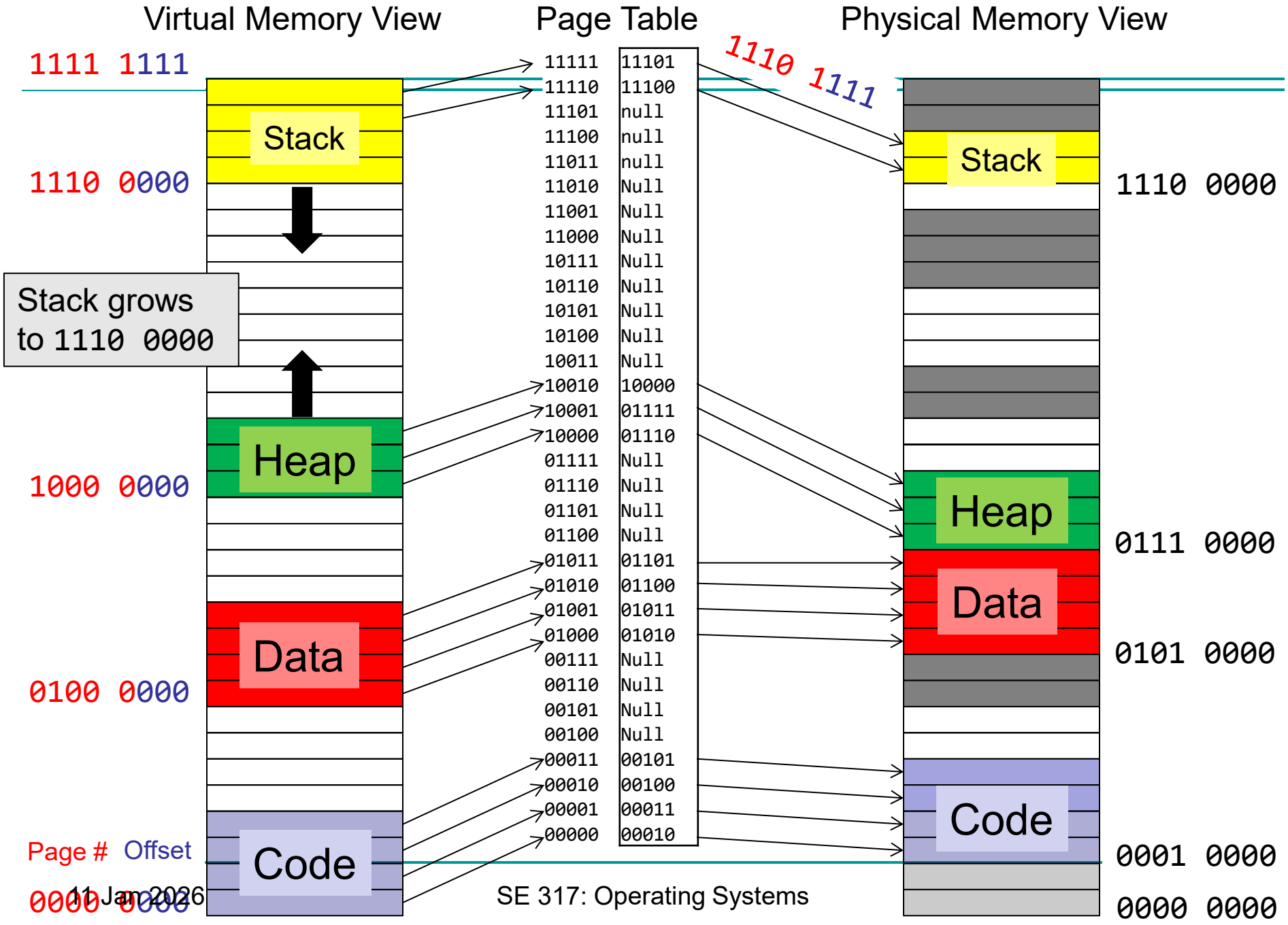
Virtual Address 2

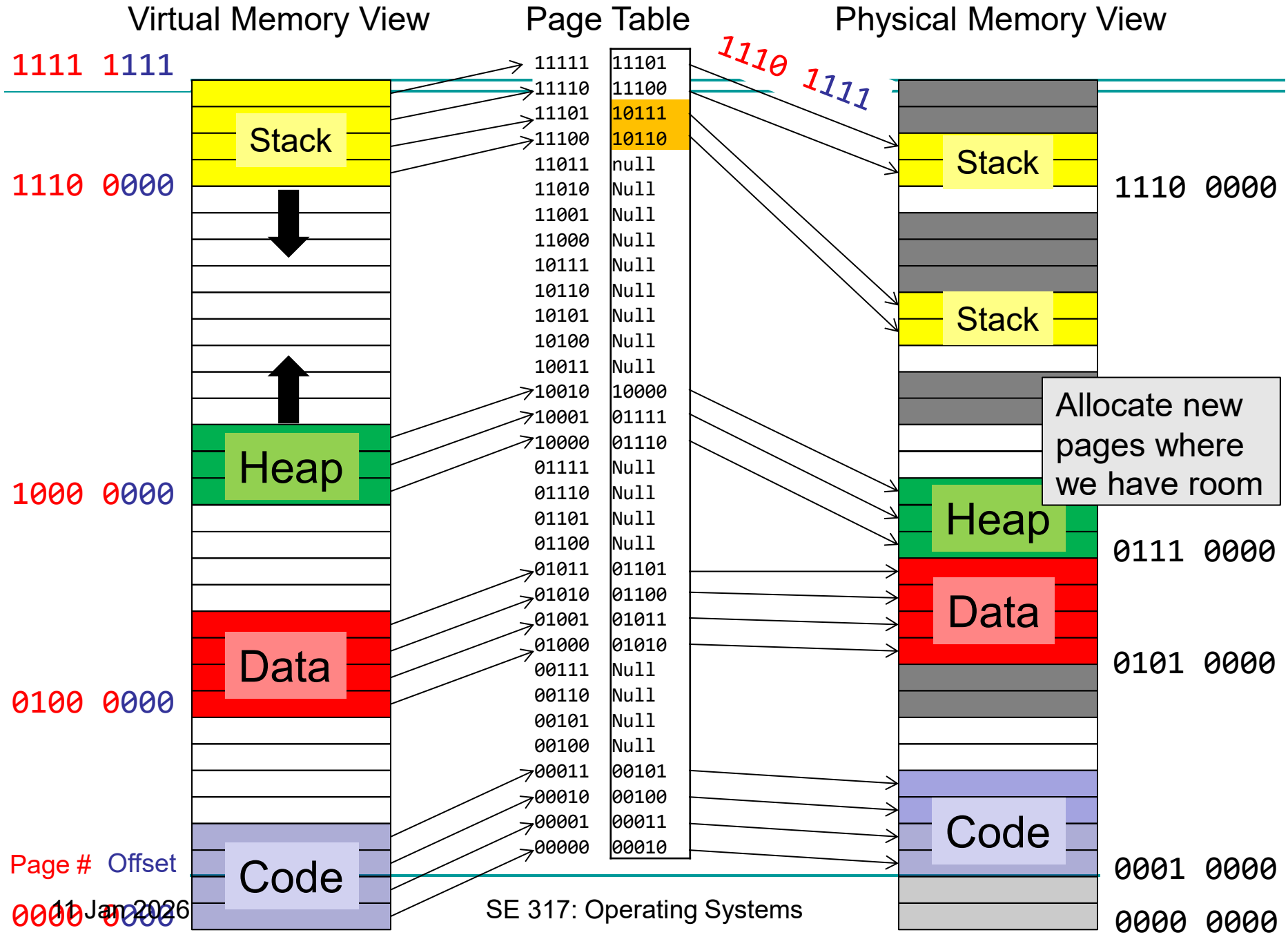


What about Sharing?




Virtual Memory View
Page Table
Physical Memory View
1111 1111
1111 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
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
1110 1111
Stack
Heap
Data
Code
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000
Page # Offset
11 Jan 2026
SE 317: Operating Systems


Virtual Memory View
Page Table
Physical Memory View
1111 1111
1110 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
Stack grows to 1110 0000
Page # Offset
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
1110 1111
Stack
Heap
Data
Code
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000


Virtual Memory View
Page Table
Physical Memory View
1111 1111
1110 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
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
1110 1111
Stack
Stack
Heap
Data
Code
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000
Allocate new pages where we have room
Page # Offset

Challenge: Table size equal to the total number of pages in virtual memory

Image source: https://www.walldevil.com/2261-artwork-chairs-giant-tables.html

Page Table Discussion

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

- 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 is really big?
    - Not all pages used all the time $\Rightarrow$ would be nice to have a working set of page table in memory
- How about combining paging and segmentation?
  - Segments with pages inside them?
  - Need some sort of multi-level translation

What is in a Page Table Entry?

- 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 in 10, 10, 12-bit offset format
  - 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

What is in a Page Table Entry?

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

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 (load it?)
  - 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

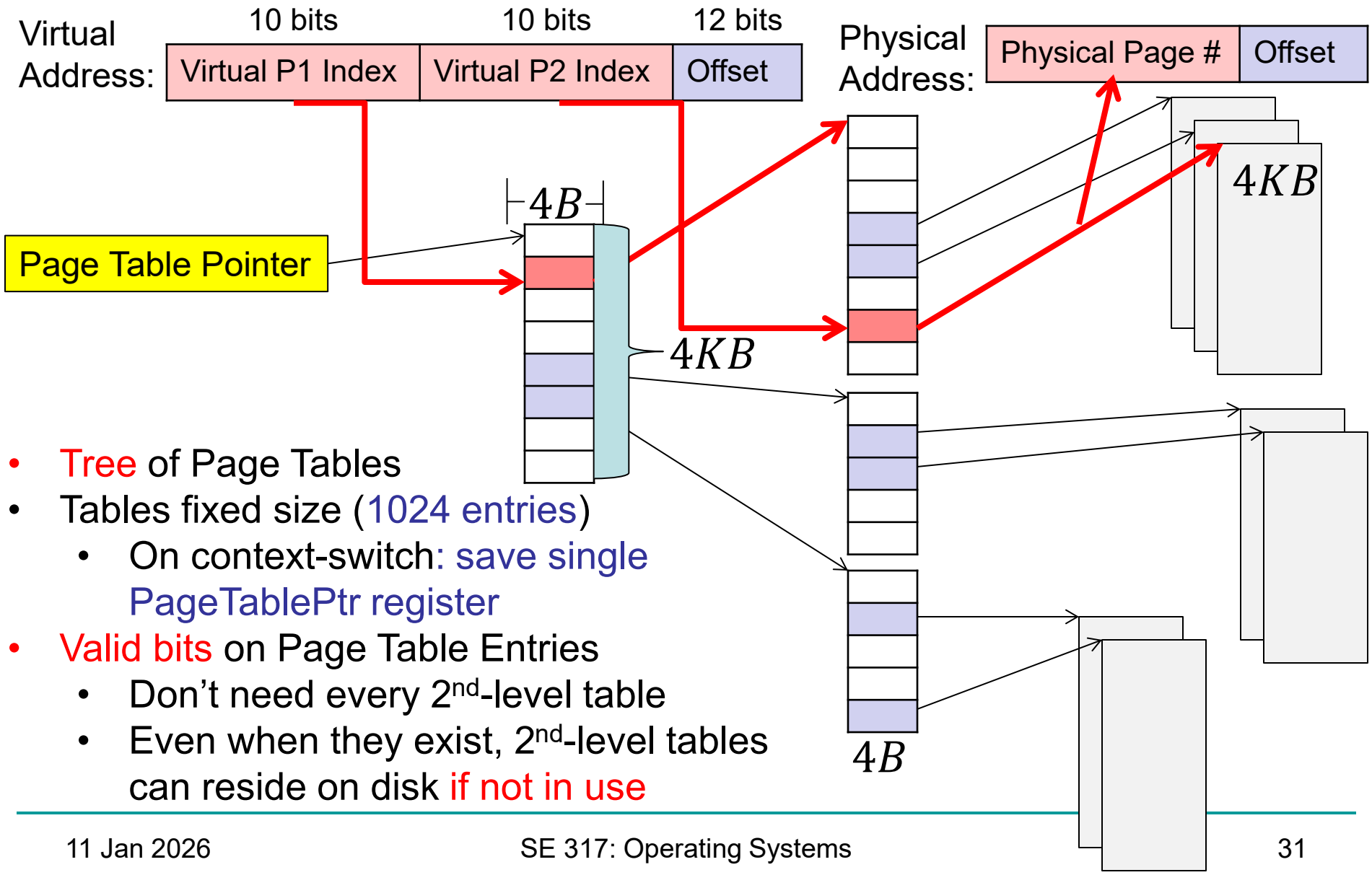
Examples of how to use a PTE

- 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

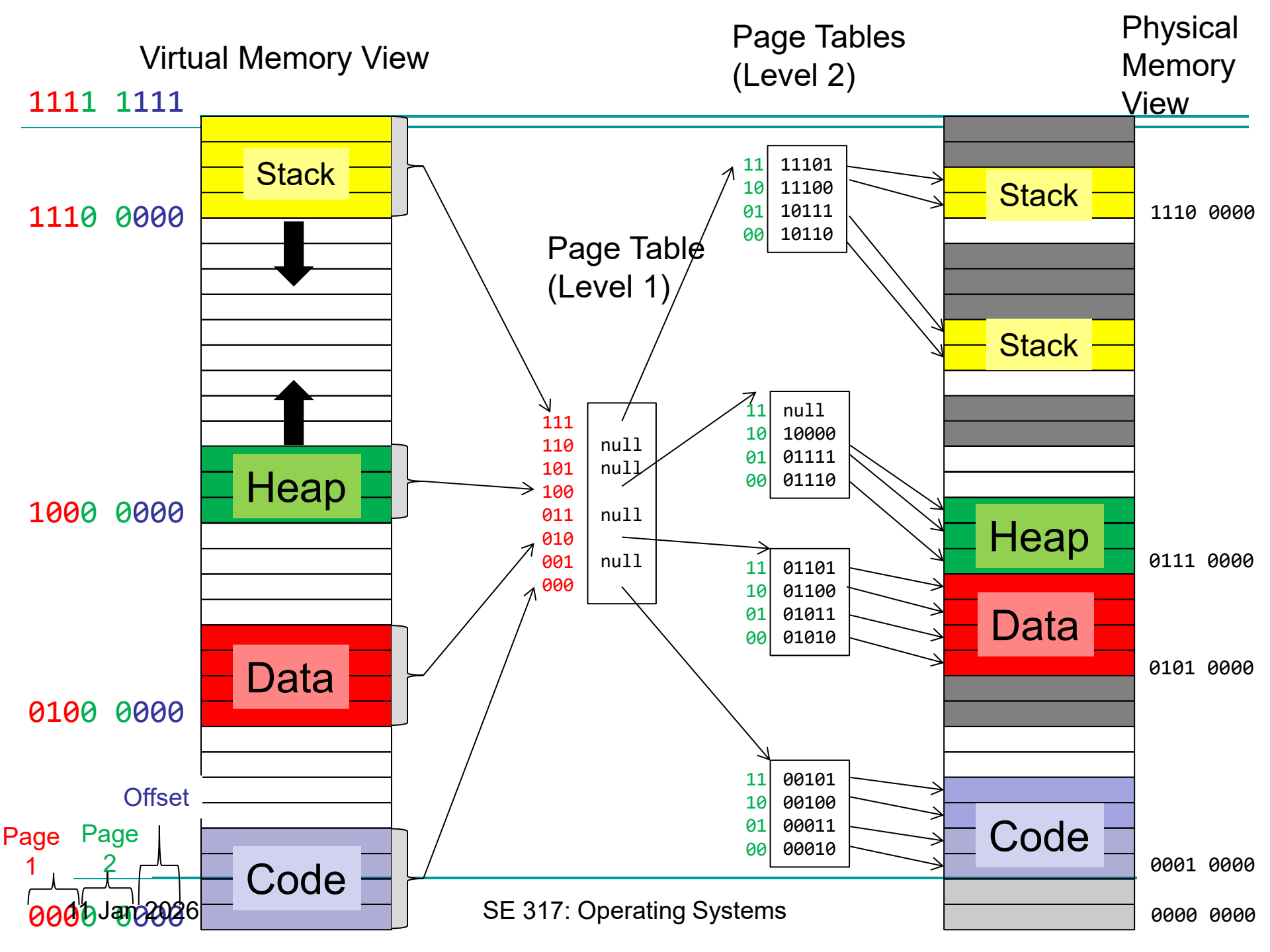
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

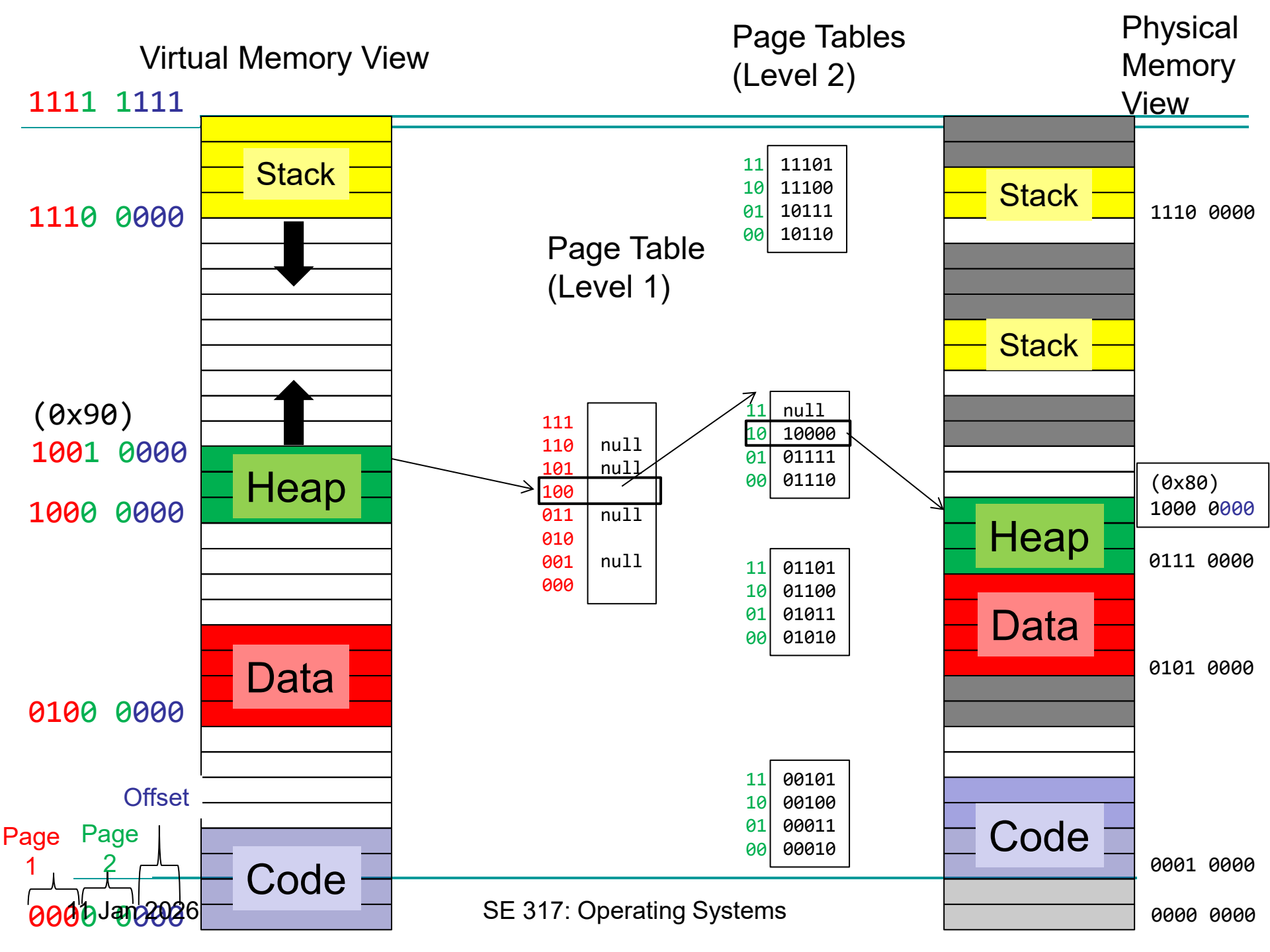
How can we make the page table closer to the memory space we actually need?

Fix for sparse address space: The 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 they exist, 2nd-level tables can reside on disk if not in use

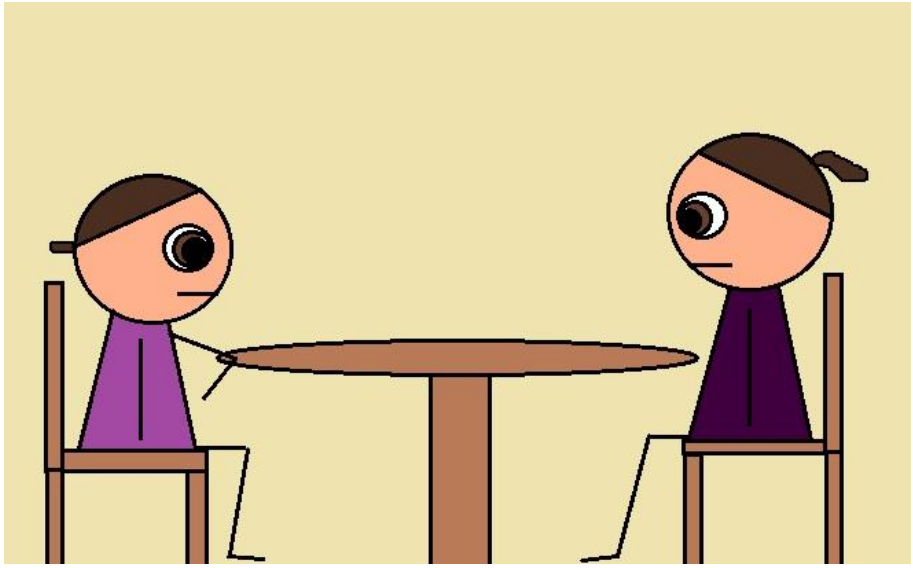

Virtual Memory View
Page Tables (Level 2)
Physical Memory View
1111 1111
1110 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
Offset
Page 1
Page 2
Page Table (Level 1)
111
110
101
100
011
010
001
000
null
null
null
null
11 11101
10 11100
01 10111
00 10110
11 null
10 10000
01 01111
00 01110
11 01101
10 01100
01 01011
00 01010
11 00101
10 00100
01 00011
00 00010
Stack
Stack
Heap
Data
Code
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000


Virtual Memory View
Page Tables (Level 2)
Physical Memory View
1111 1111
Stack
1110 0000
Page Table (Level 1)
11 11101
10 11100
01 10111
00 10110
(0x90)
1001 0000
Heap
1000 0000
111
110 null
101 null
100
011 null
010
001 null
000
11 null
10 10000
01 01111
00 01110
Stack
1110 0000
(0x80)
1000 0000
Heap
0111 0000
Data
0101 0000
Data
0100 0000
11 01101
10 01100
01 01011
00 01010
Offset
Page 1
Page 2
11 00101
10 00100
01 00011
00 00010
Code
0001 0000
Code
0000 0000
0000 0000
11 Jan 2026
SE 317: Operating Systems

Result

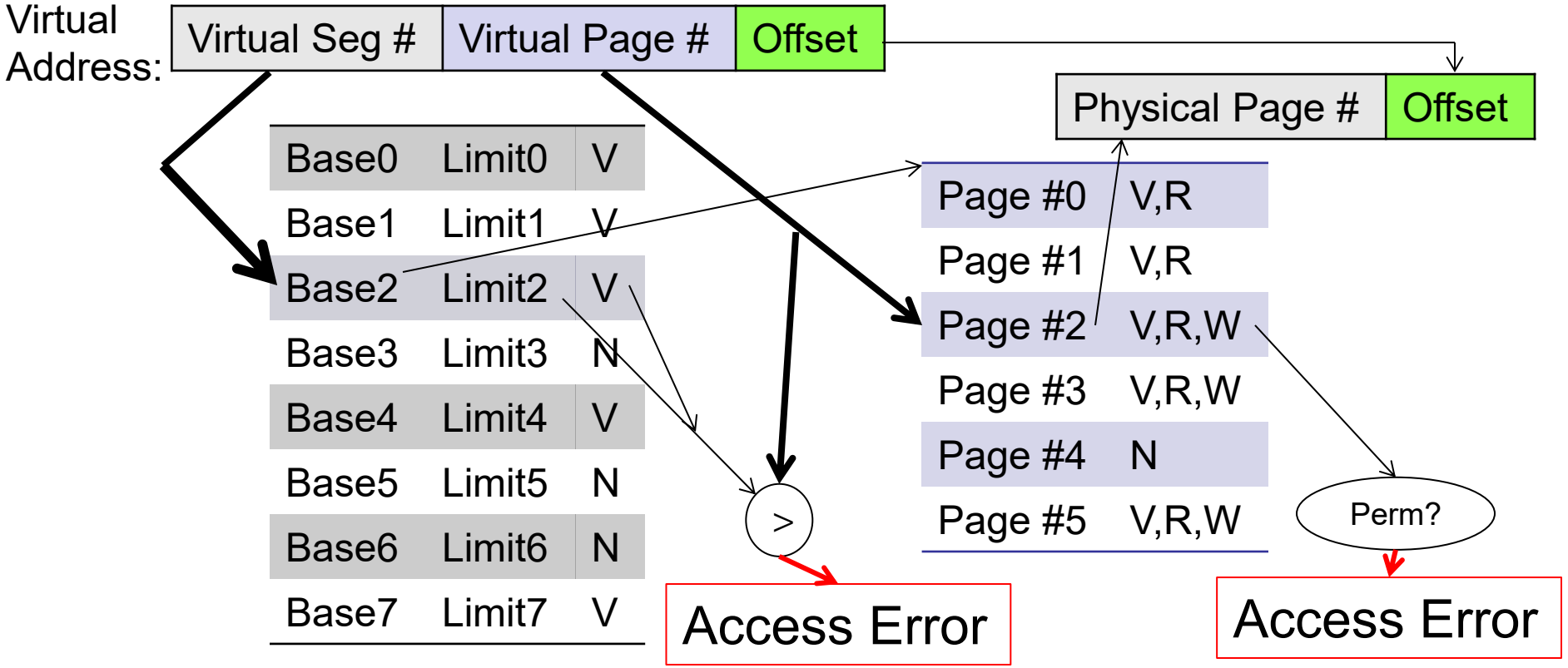
In best case, total size of page tables ≈ number of pages used by program virtual memory.

Requires two additional memory access!



Multi-level Translation: Segments + Pages

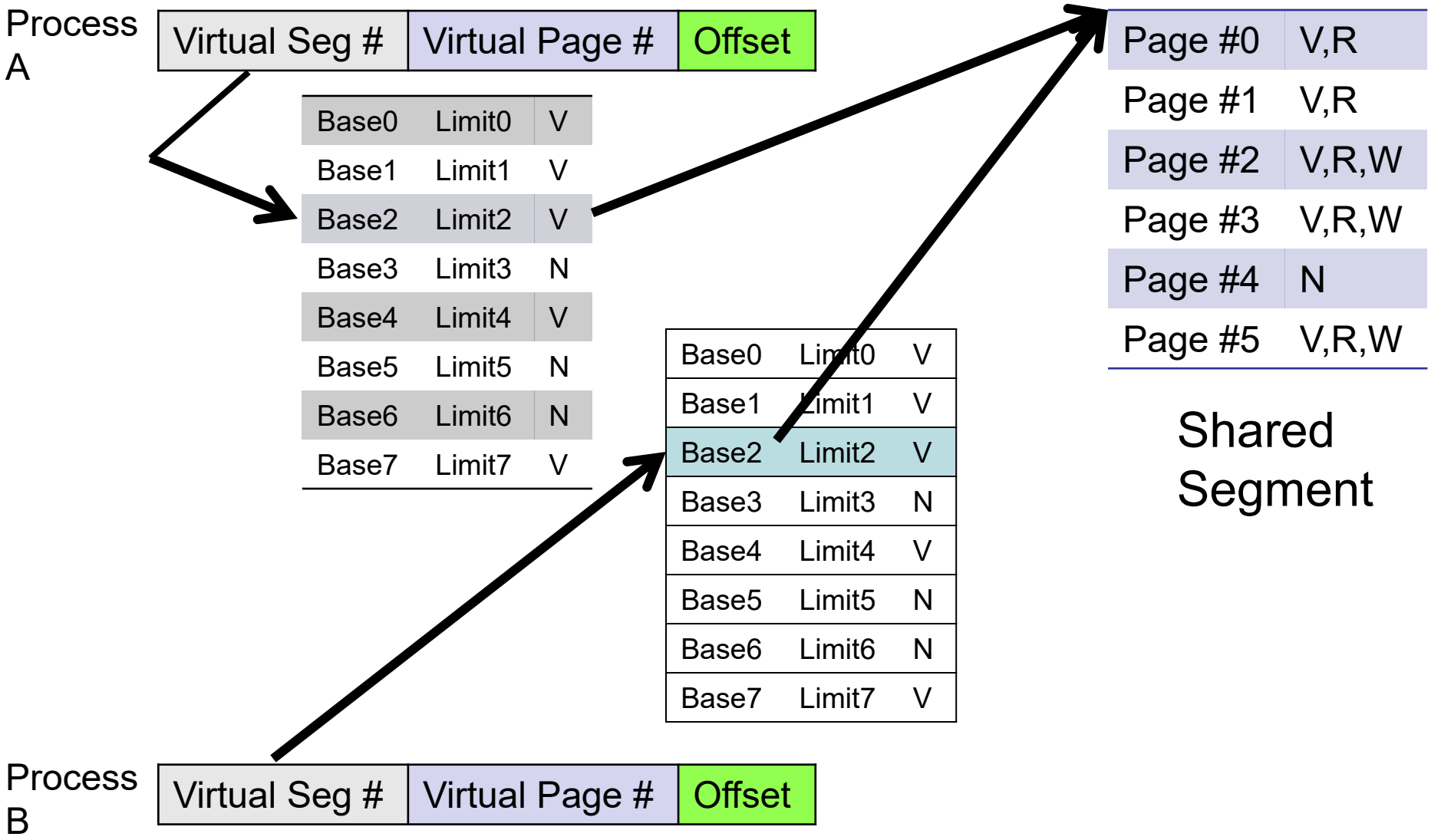
- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



Multi-level Translation: Segments + Pages

- 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)?

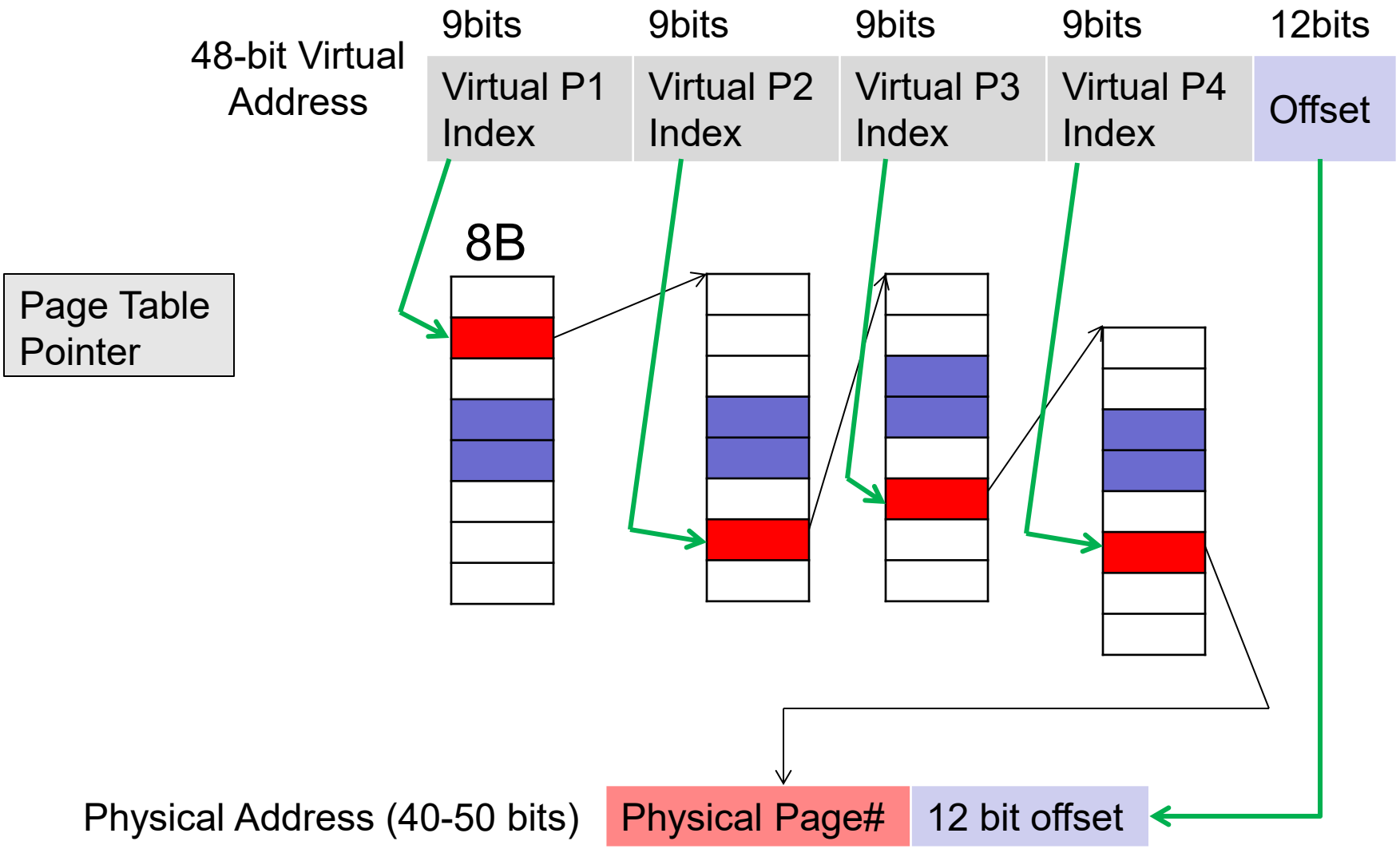


Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - Therefore sparse address spaces are easy
  - Easy memory allocation
  - Easy sharing
    - Share at segment or page level (need additional reference counting)

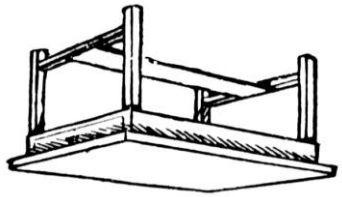
- Cons:
  - One pointer per page (typically $4K$ – $16K$ pages today)
  - Page tables need to be contiguous
    - However, previous example keeps tables to exactly one page in size
  - Two (or more, if > 2 levels) lookups per reference
    - Seems very expensive!

X86_64: Four-level page table!



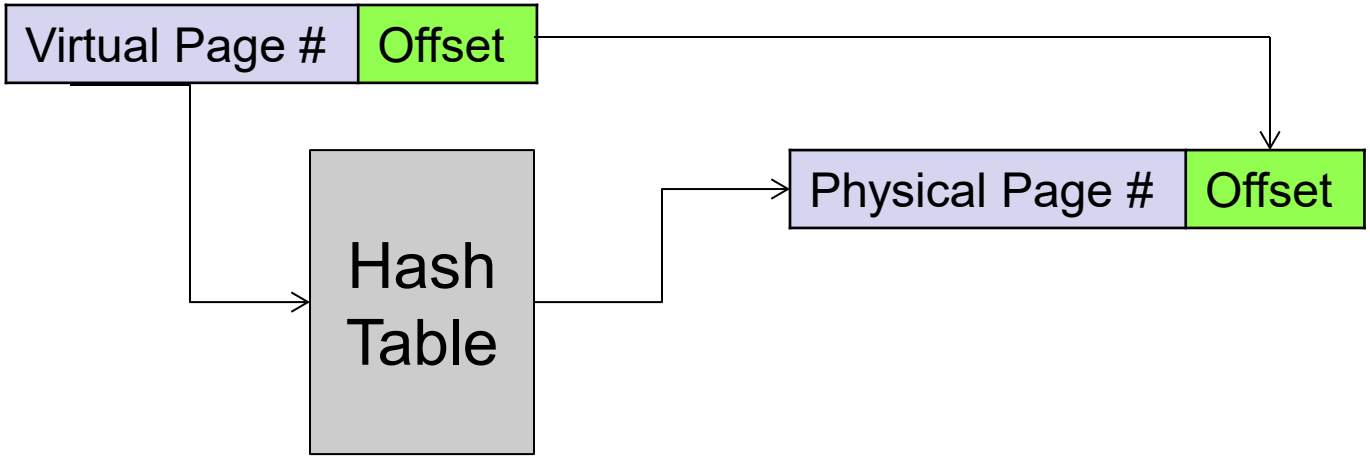
Any other ideas to make it smaller?

Inverted Page Table



- 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

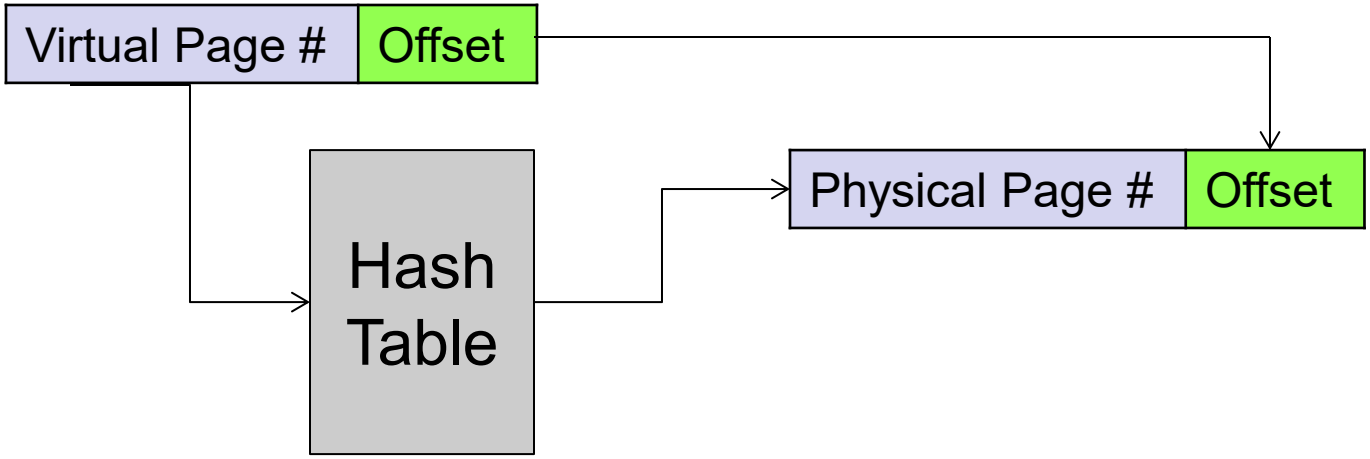
- Answer: use a hash table
  - Called an "Inverted Page Table"



Inverted Page Table

- Pros:
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces

- Cons: Complexity of managing hash changes
  - Often in hardware!



IA64: 64bit addresses: Six-level page table?!?

64 bit Virtual Address	7bits	9bits	9bits	9bits	9bits	9bits	12bits
	Virtual P1 Index	Virtual P2 Index	Virtual P3 Index	Virtual P4 Index	Virtual P5 Index	Virtual P6 Index	Offset

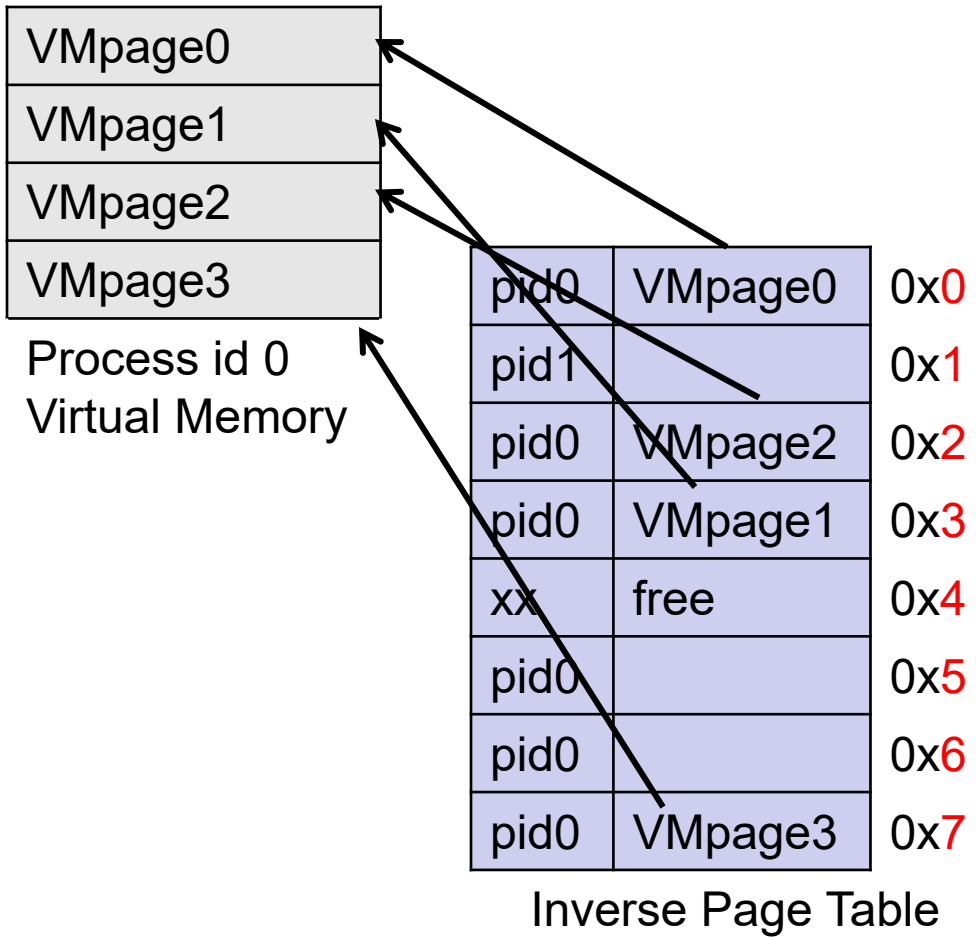
No!

Too Slow!

Too many almost empty tables

IA64: Inverse Page Table (IPT)

Idea: Index page table by physical pages instead of VM



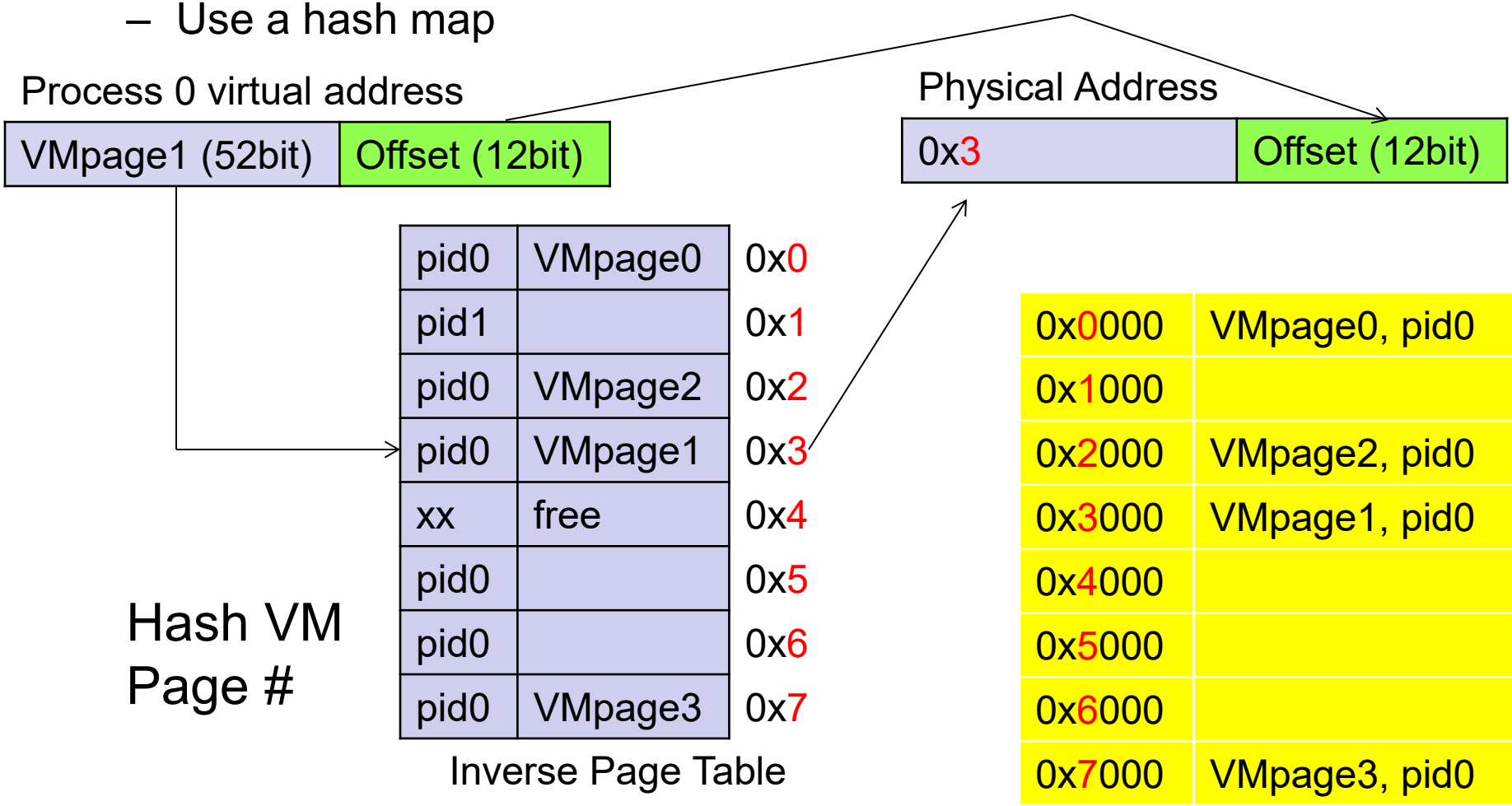
0x0000	VMpage0, pid0
0x1000	
0x2000	VMpage2, pid0
0x3000	VMpage1, pid0
0x4000	
0x5000	
0x6000	
0x7000	VMpage3, pid0

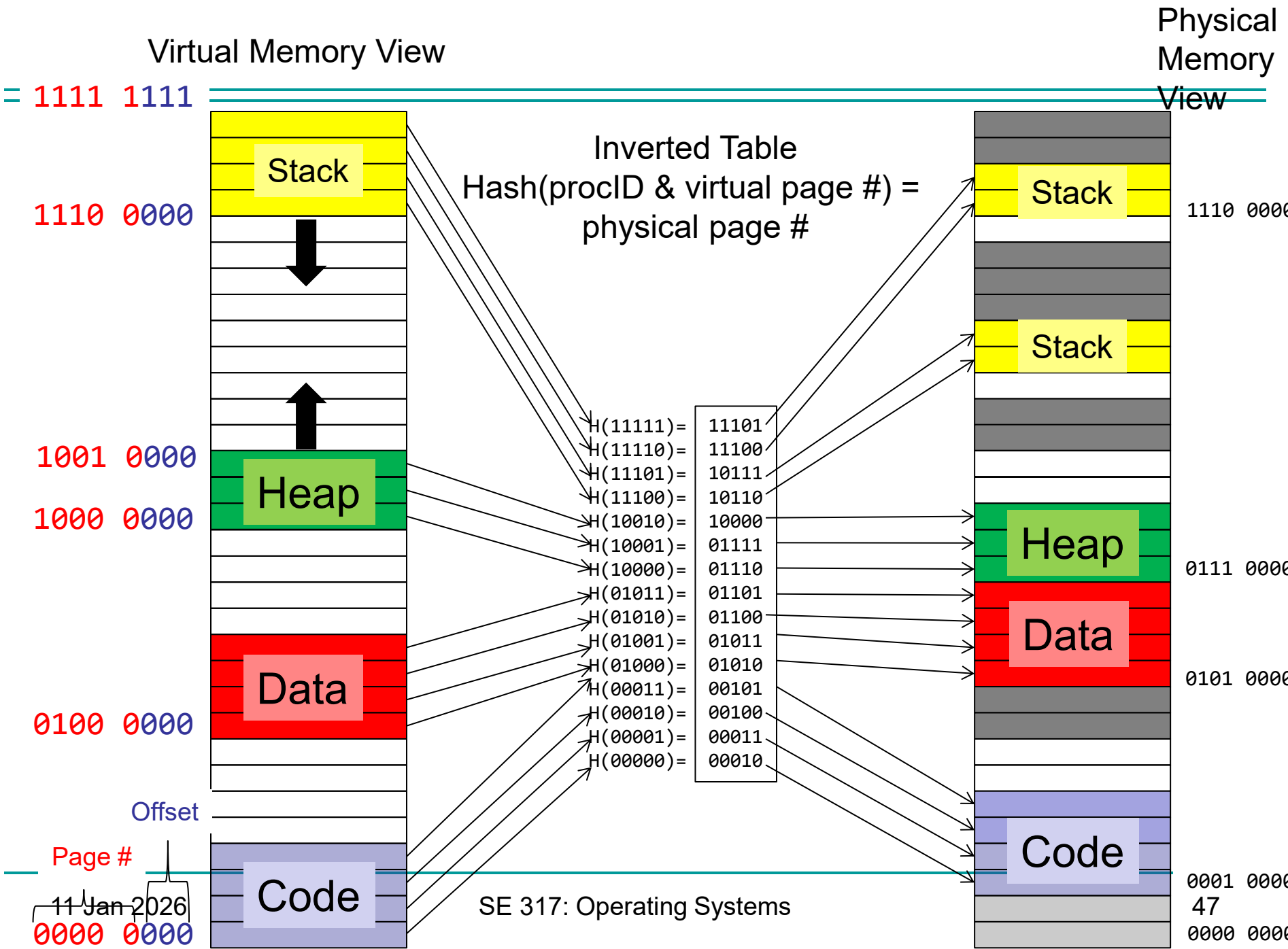
Physical Memory in 4KB pages

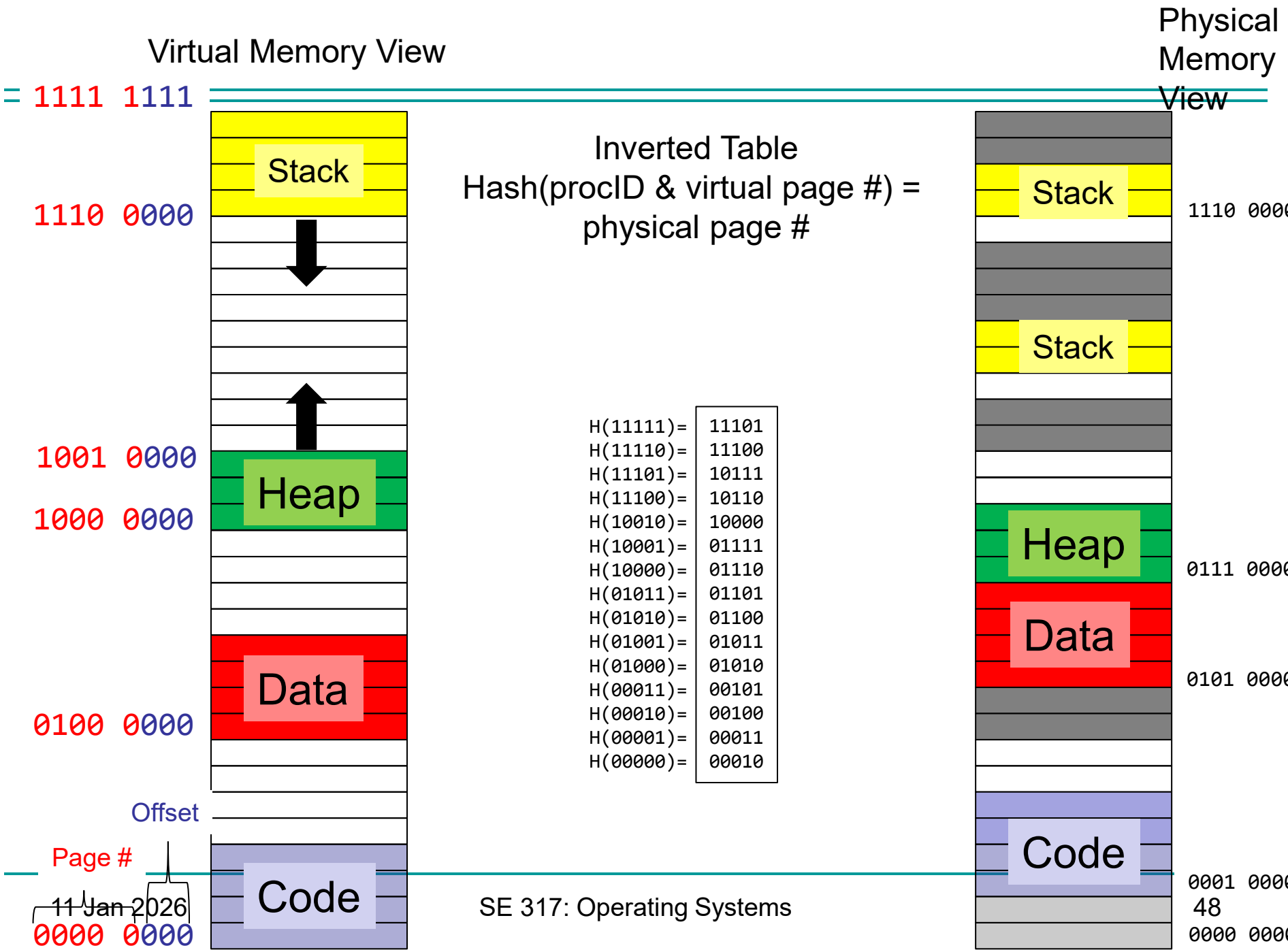
Page numbers in red

IPT address translation

- Need an associative map from VM page to IPT address:
  - Use a hash map



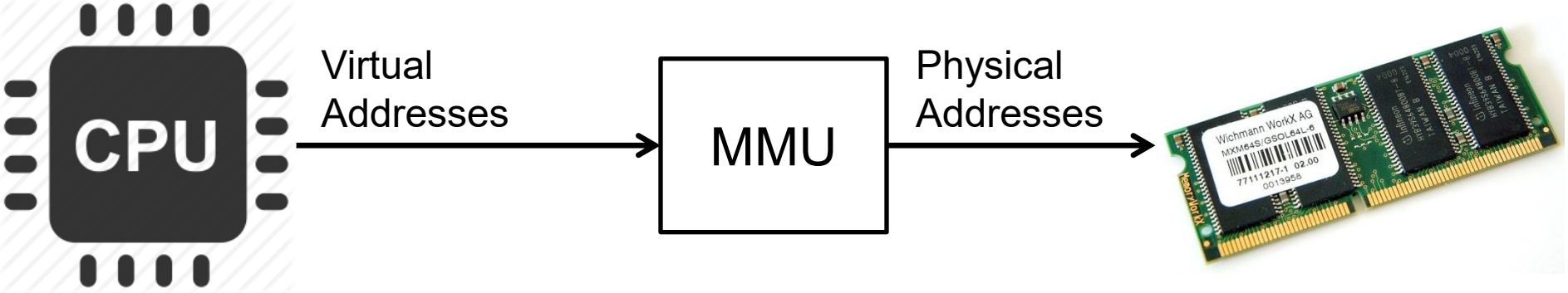

Virtual Memory View
Physical Memory View
1111 1111
1110 0000
1001 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
Offset
Page #
Inverted Table
Hash(procID & virtual page #) = physical page #
H(11111)= 11101
H(11110)= 11100
H(11101)= 10111
H(11100)= 10110
H(10010)= 10000
H(10001)= 01111
H(10000)= 01110
H(01011)= 01101
H(01010)= 01100
H(01001)= 01011
H(01000)= 01010
H(00011)= 00101
H(00010)= 00100
H(00001)= 00011
H(00000)= 00010
Stack
Stack
Heap
Data
Code
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000


Virtual Memory View
Physical Memory View
1111 1111
1110 0000
1001 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
Offset
Page #
Inverted Table
Hash(procID & virtual page #) = physical page #
H(11111)= 11101
H(11110)= 11100
H(11101)= 10111
H(11100)= 10110
H(10010)= 10000
H(10001)= 01111
H(10000)= 01110
H(01011)= 01101
H(01010)= 01100
H(01001)= 01011
H(01000)= 01010
H(00011)= 00101
H(00010)= 00100
H(00001)= 00011
H(00000)= 00010
Stack
Stack
Heap
Data
Code
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000

Address Translation Comparison

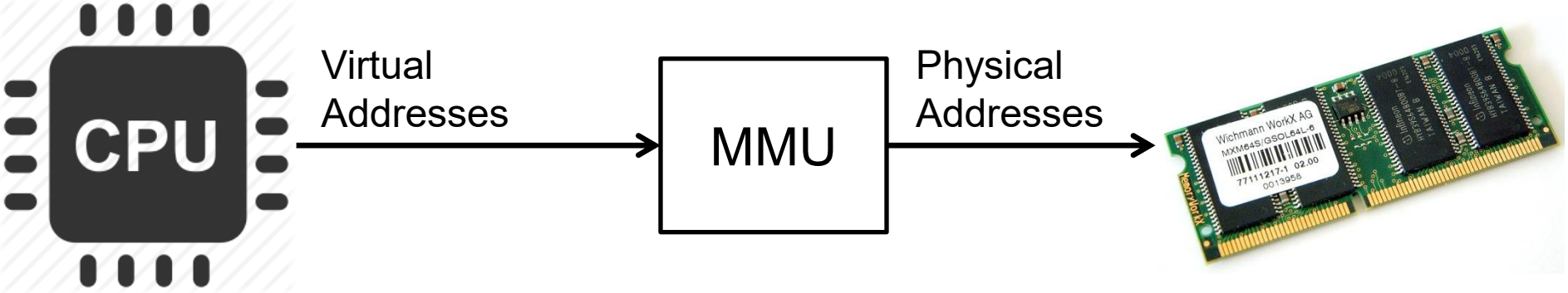
	Advantages	Disadvantages
Simple Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation	Table size ~ # of pages 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

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 later
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware

How is the translation accomplished?



- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!

- In fact, need way to *cache* translations for either case!

Conclusion

- Address Spaces and Segmentation
  - Fragmentation
- Page tables