Lecture 7
Paging

Prof. Yinqian Zhang

Summer 2022

Outline

- Introduction to paging
- Multi-level page tables
- Other page table structures
- Real-world paging schemes
- Translation lookaside buffer

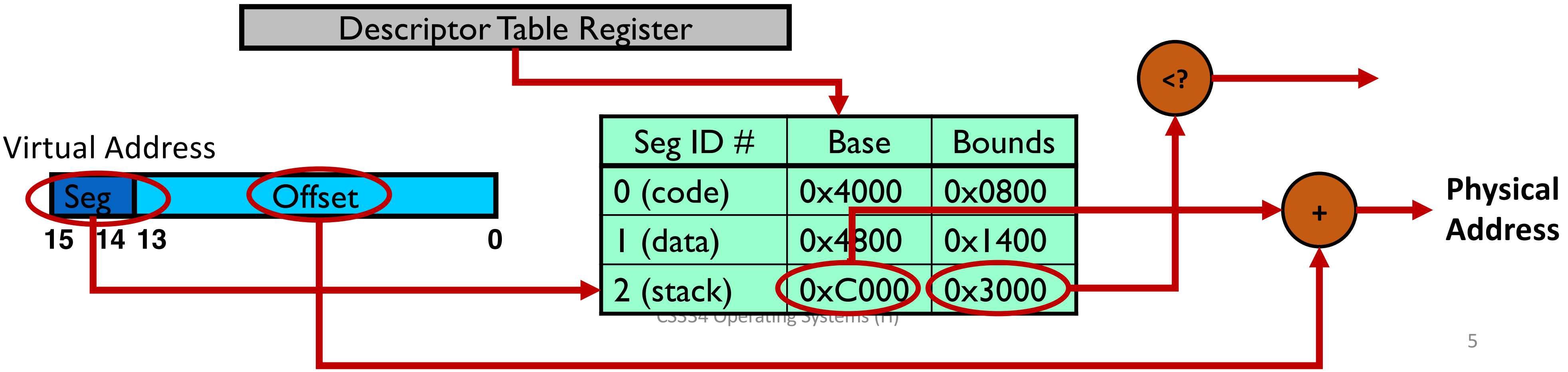
Introduction to Paging

Recall: Problems with Segmentation

- OS context switch must also save and restore all pairs of segment registers
- A segment may grow, which may or may not be possible
- Management of free spaces of physical memory with variable-sized segments
- **External fragmentation**: gaps between allocated segments
  - Segmentation may also have internal fragmentation if more space allocated than needed.

Solutions for Segmentation

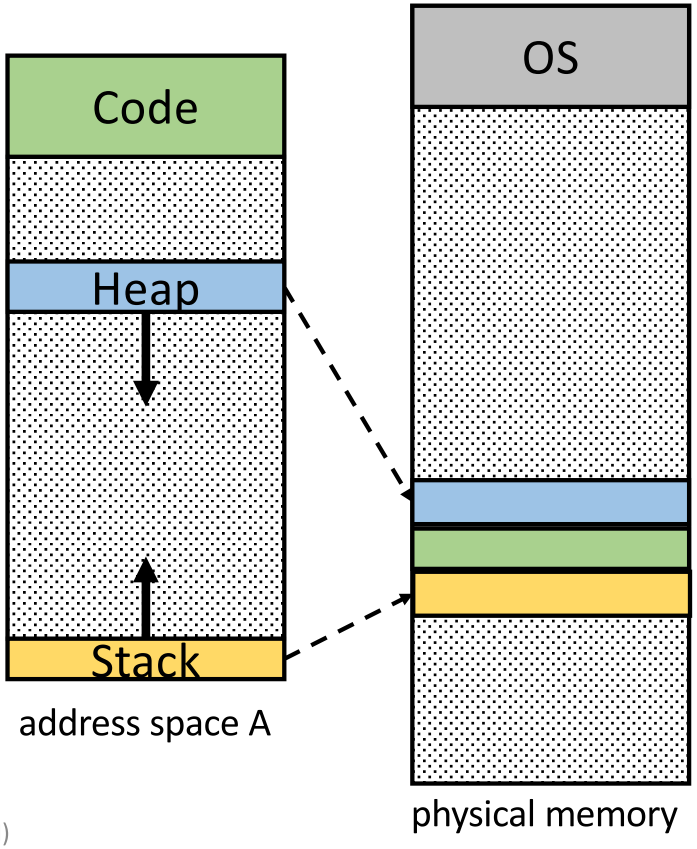
- OS context switch must also save and restore all pairs of segment registers
  - **Table of base/bounds stored in memory** rather than registers
  - Pointer to the table stored in a register
  - **Cons:** one more memory read per address translation



Seg ID #	Base	Bounds
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (stack)	0xC000	0x3000

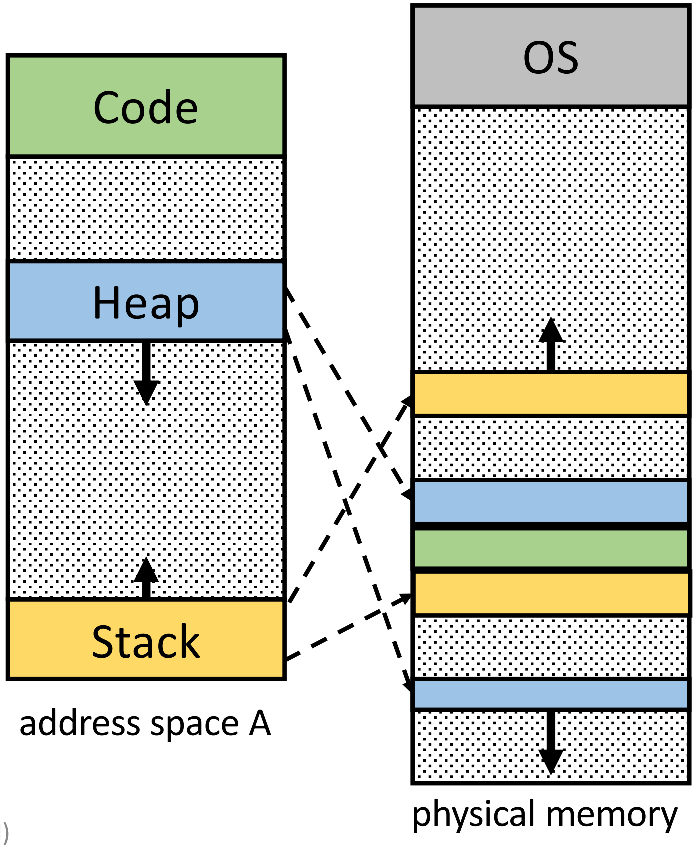
Solutions for Segmentation (Cont'd)

- A segment may grow, which may or may not be possible



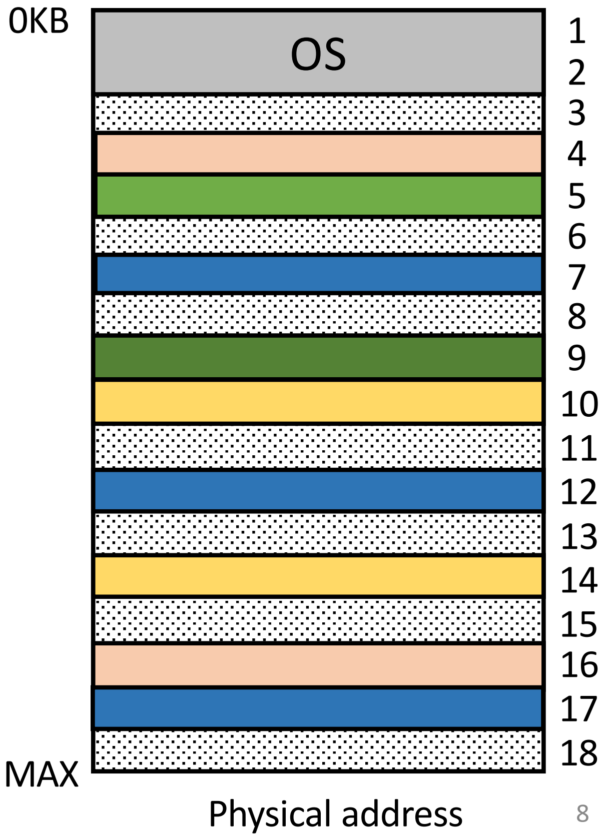
Solutions for Segmentation (Cont'd)

- A segment may grow, which may or may not be possible
  - **Virtually continuous memory can be physically discontinuous**



Solutions for Segmentation (Cont'd)

- Management of free spaces of physical memory with variable-sized segments
  - **Fixed-sized segments**
  - Physical memory curved into **fixed** sized chunks, **index by an integer**
  - Can use simple vector of bits to handle allocation (1-allocated, 0-free)
    110110101101010110
- External fragmentation
  - **Fixed-sized segments**
  - Each request is of the same fixed size – always fit, no external fragmentation



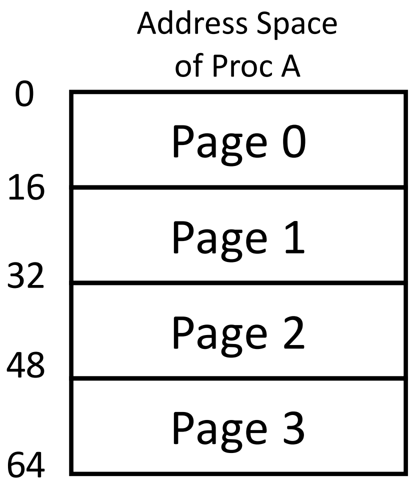
Paging: Put All These Ideas Together

- Physical memory conceptually divided into fixed size
  - Each is called a **page frame**
  - Typical size 1KB to 16KB, most ISA uses 4KB
- Virtual address space conceptually divided into the same size
  - Each is called a **page**
- Page mapped to page frame
  - One to one mapping
  - Many to one mapping -> memory sharing
- One page table per process
  - Resides in physical memory
  - One entry for one virtual->physical translation

How big is one page (frame)?

- Too big -> internal fragmentation
- Too small -> page table too big

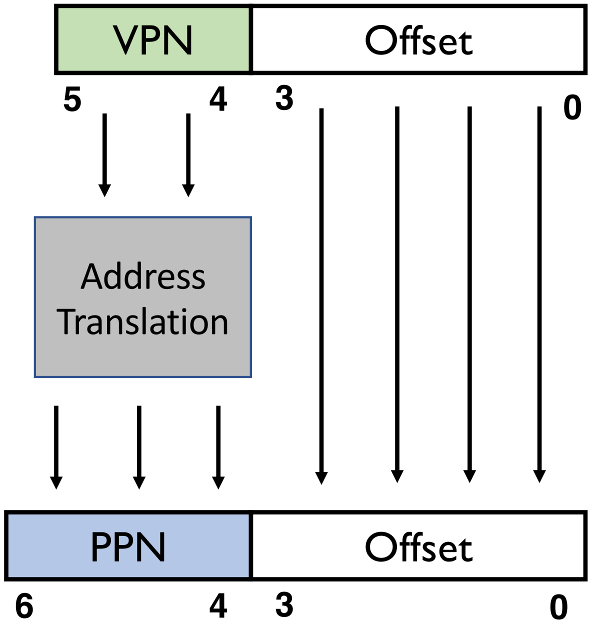
Paging Illustrated



Address Space of Proc A

0 Page 0
16 Page 1
32 Page 2
48 Page 3
64

Virtual Address = Virtual Page Number || Offset

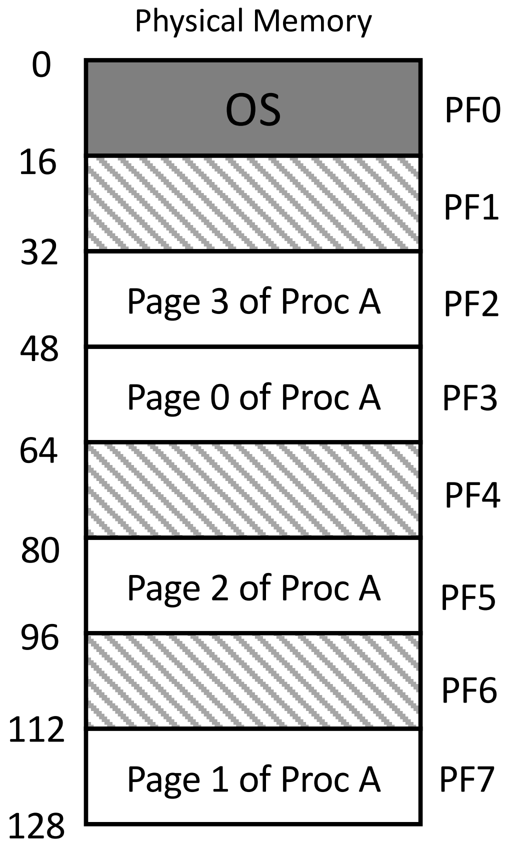


VPN | Offset
5 4 3 0

Address Translation

PPN | Offset
6 4 3 0

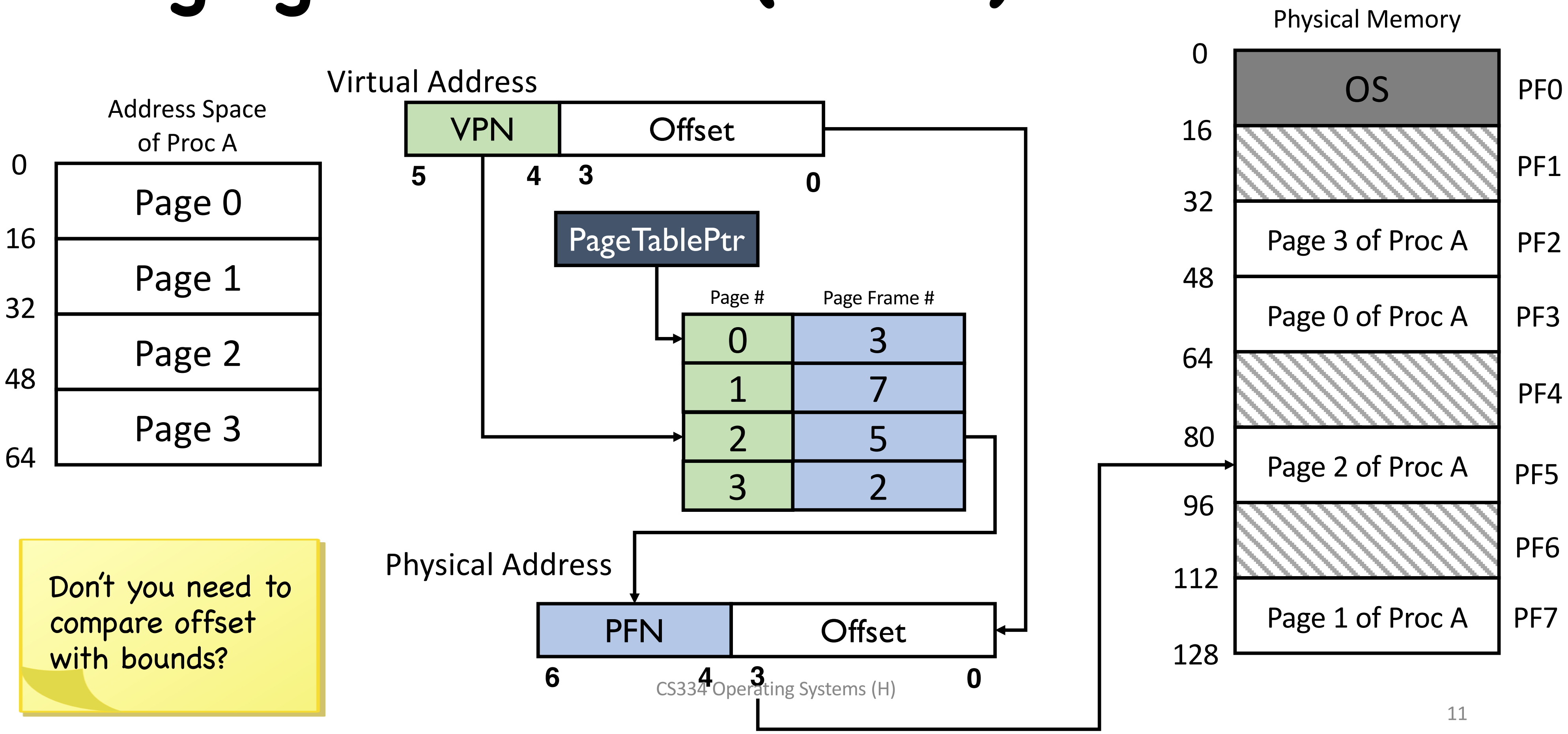
Physical Address = Physical Page Number || Offset



Physical Memory

0 OS PF0
16 PF1
32 Page 3 of Proc A PF2
48 Page 0 of Proc A PF3
64 PF4
80 Page 2 of Proc A PF5
96 PF6
112 Page 1 of Proc A PF7
128

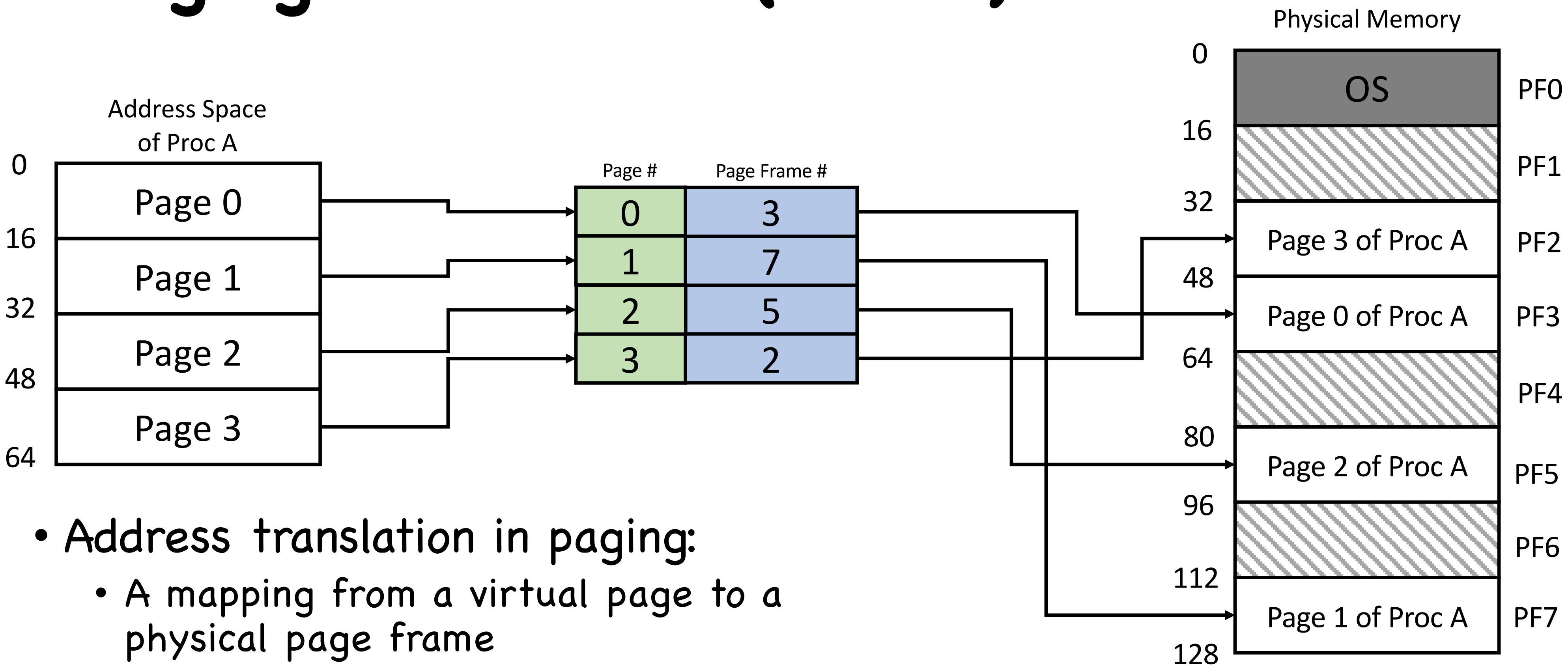
Paging Illustrated (Cont'd)



Page #	Page Frame #
0	3
1	7
2	5
3	2

Don't you need to compare offset with bounds?

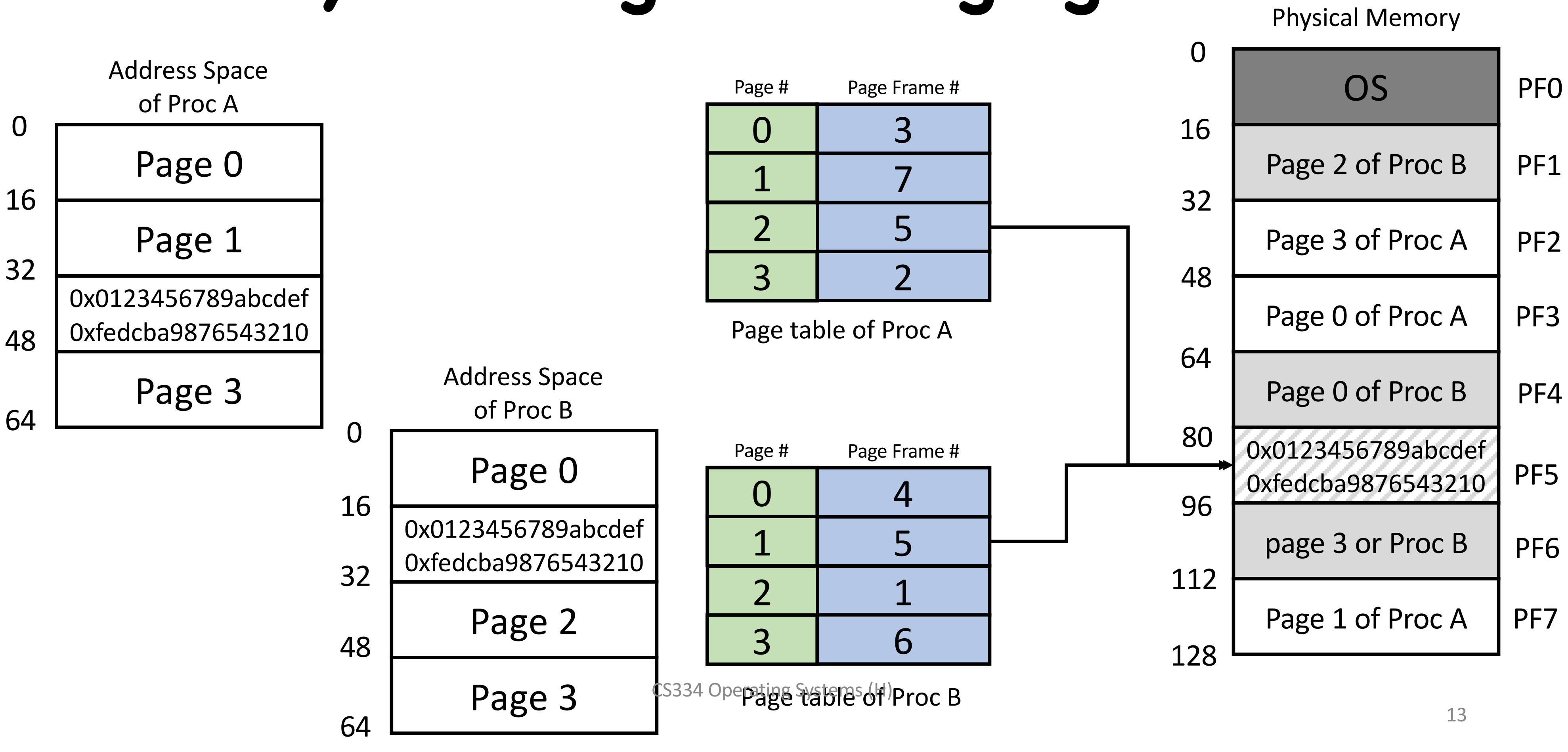
Paging Illustrated (Cont'd)



Page #	Page Frame #
0	3
1	7
2	5
3	2

- Address translation in paging:
  - A mapping from a virtual page to a physical page frame

Memory Sharing with Paging



Address Space of Proc A

0	Page 0
16	Page 1
32	0x0123456789abcdef 0xfedcba9876543210
48	Page 3
64	

Address Space of Proc B

0	Page 0
16	0x0123456789abcdef 0xfedcba9876543210
32	Page 2
48	Page 3
64	

Page #	Page Frame #
0	3
1	7
2	5
3	2

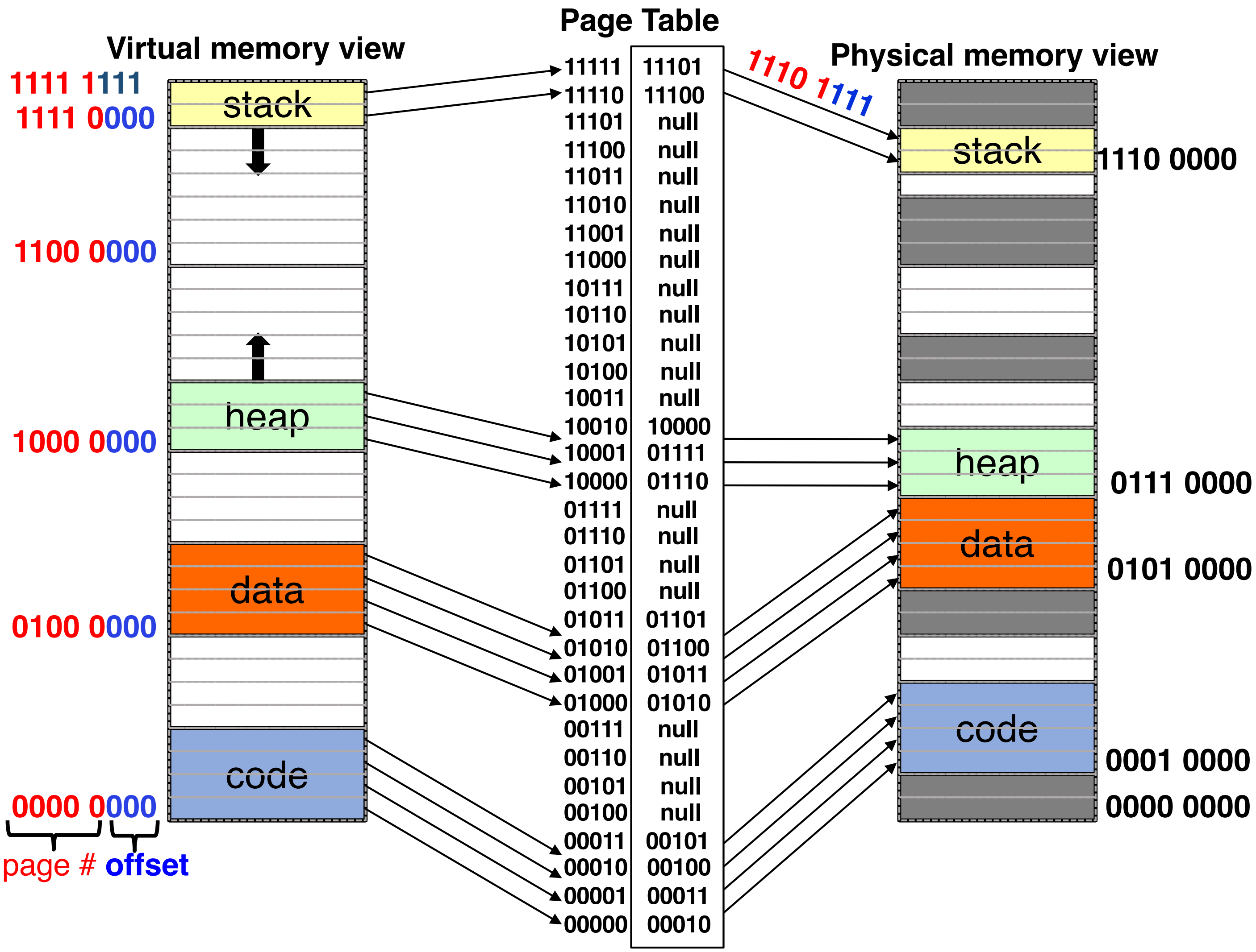
Page table of Proc A

Page #	Page Frame #
0	4
1	5
2	1
3	6

Page table of Proc B

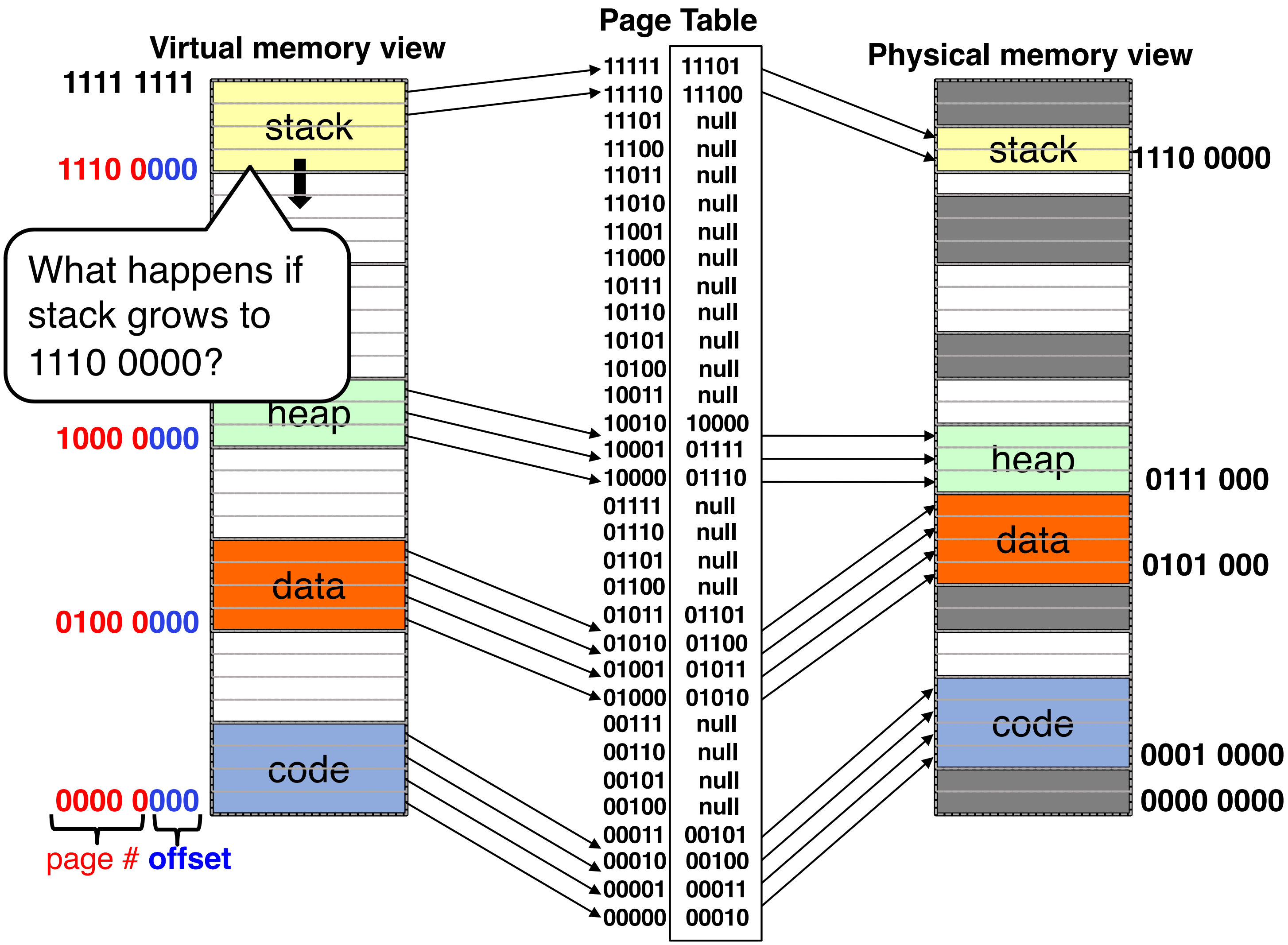
Physical Memory

0	OS	PF0
16	Page 2 of Proc B	PF1
32	Page 3 of Proc A	PF2
48	Page 0 of Proc A	PF3
64	Page 0 of Proc B	PF4
80	0x0123456789abcdef 0xfedcba9876543210	PF5
96	page 3 or Proc B	PF6
112	Page 1 of Proc A	PF7
128		



Page Table

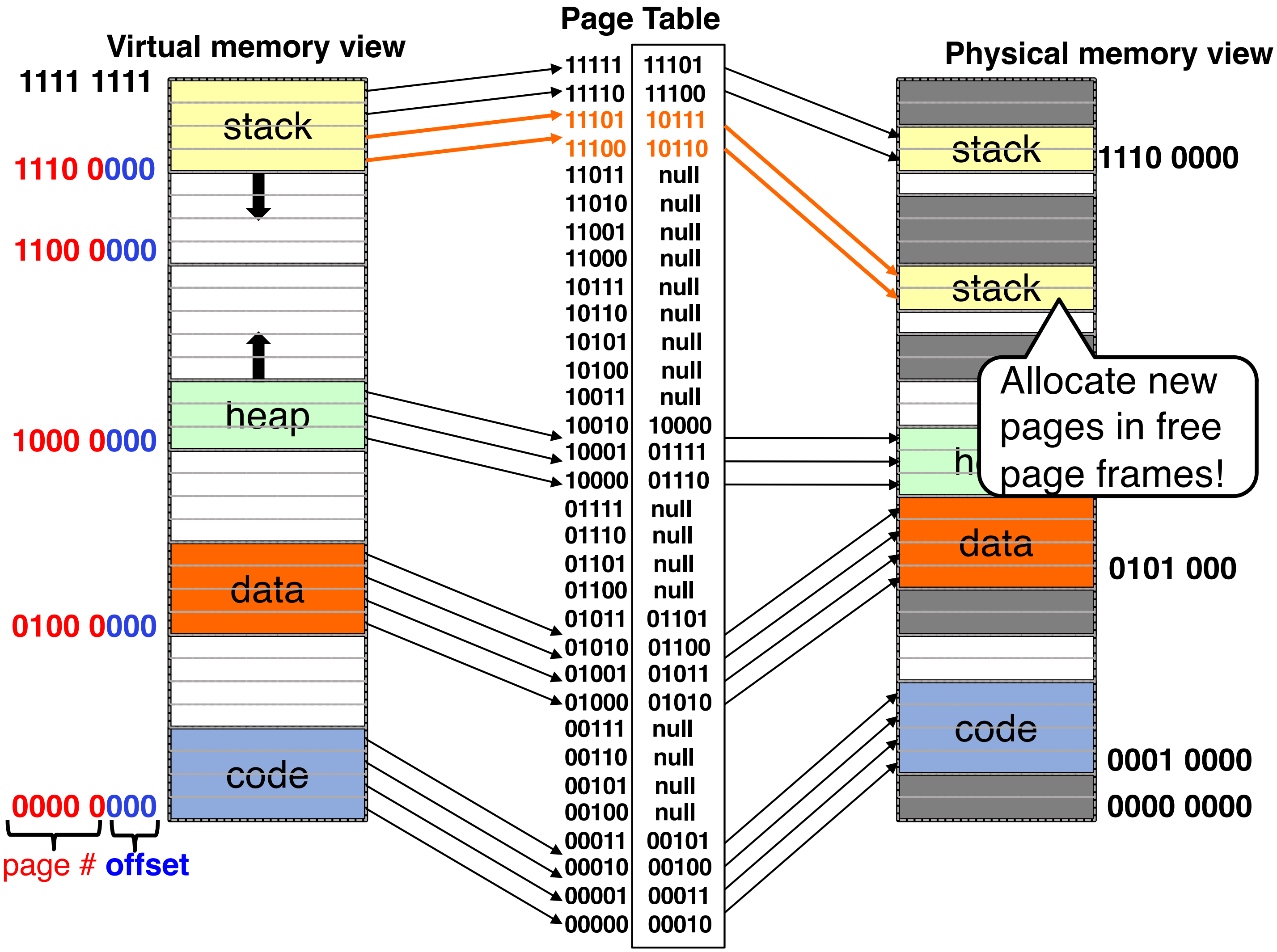
11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101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



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

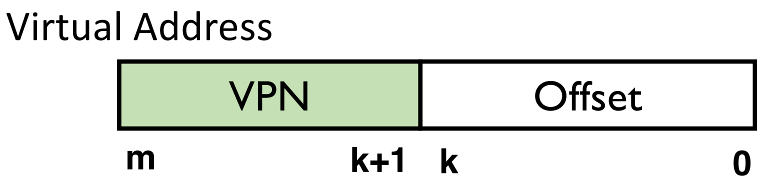


Page Table

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

Virtual Address and Paging

- Length of virtual address determines size of address space
- Length of offset determines size of a page/page frame
- In case of m-bit virtual address and k-bit offset
  - Size of address space: 2^m
  - Size of a page: 2^k
  - e.g., 32-bit virtual address, 4KB page: m = 32, k = 12



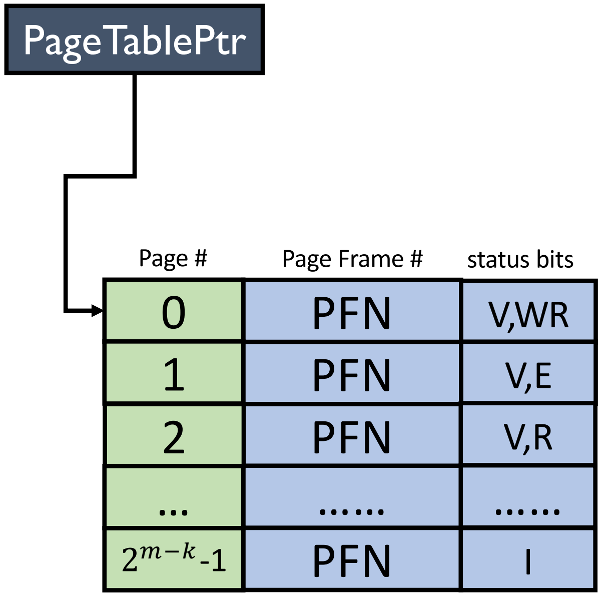
Page #	Page Frame #
0	PFN
1	PFN
2	PFN
…	……
2^{m-k}-1	PFN

Page Table Entry

- An entry in the page table is called a page table entry (PTE).
- Besides **PFN,** PTE also contains **a valid bit**
  - Virtual pages with no valid mapping: valid bit = 0
  - Important for sparse address space (e.g., 2^64 bytes)
- PTE also contains **protection bits**
  - Permission to read from or write, or execute code on this page
- PTE also contains **an access bit, a dirty bit, a present bit**
  - Present bit: whether this page is in physical memory or on disk
  - Dirty bit: whether the page has been modified since it was brought into memory
  - Access bit: whether a page has been accessed

How Big are Page Tables

- Assume 32-bit machine and 4KB pages, and each PTE takes 4 bytes (PFN plus status bits)
  - Number of PTEs: $2^{32-12} = 2^{20}$
  - Size of page table: 4MB
  - What if 100 processes running? 400MB
- Page tables are stored in memory and context switch only changes the pointer to page table (e.g., CR3 register)

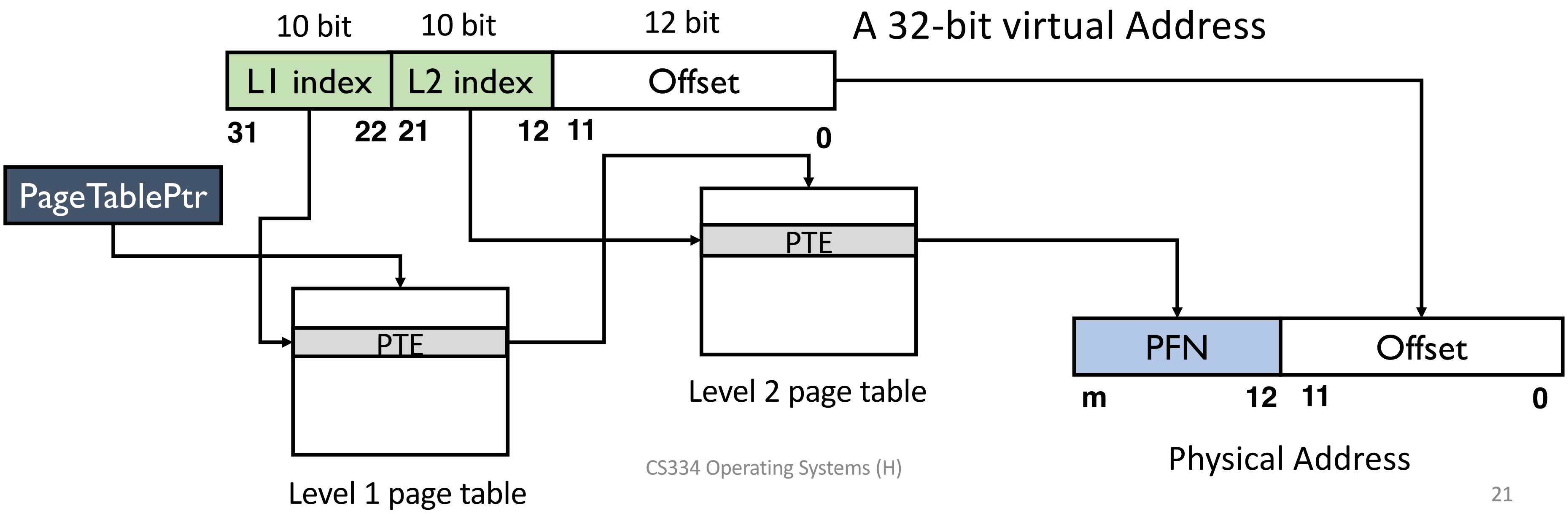


Page #	Page Frame #	status bits
0	PFN	V,WR
1	PFN	V,E
2	PFN	V,R
…	……	……
2^{m-k}-1	PFN	I

Multi-level Page Tables

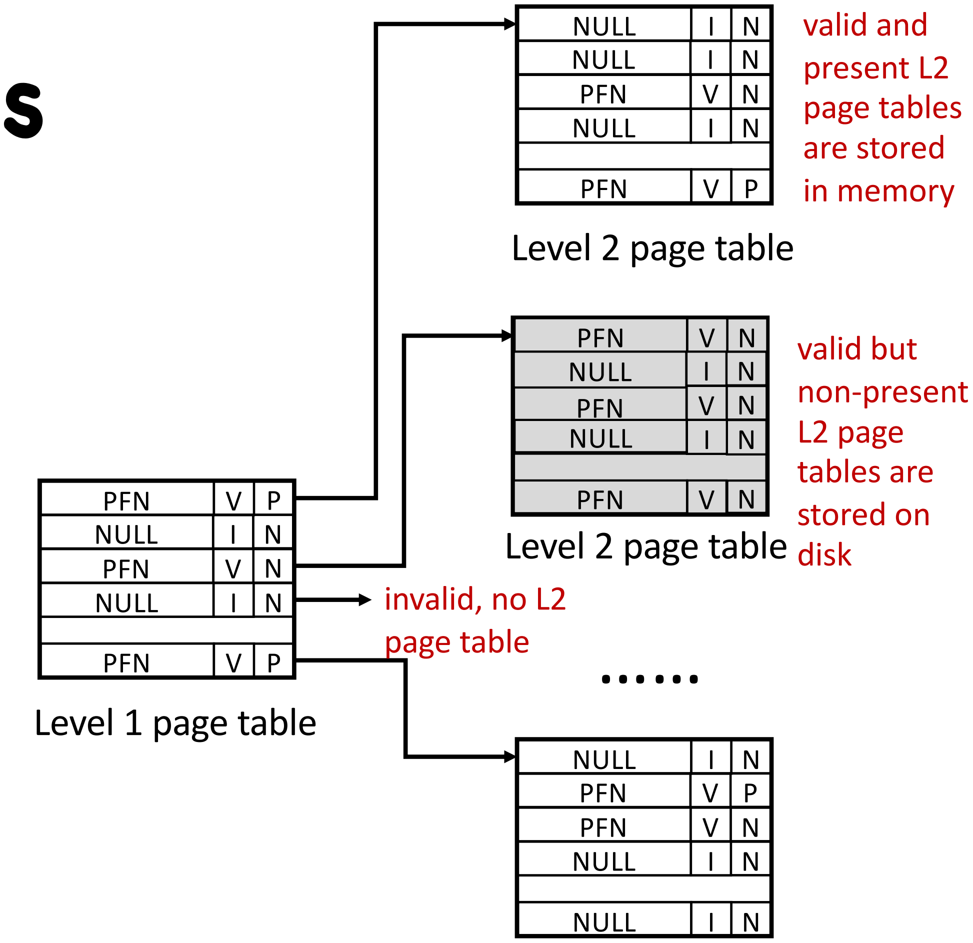
Two-level Page Tables

- Use two levels of page tables to save space



A Tree of Page Tables

- A page table also occupies a page frame
  - e.g., 4KB page and 4-byte PTE: 1024 entries per table
- Two-level page table forms a tree of page tables
  - In theory 1024 level-2 page tables, but only a subset are valid
  - A subset of valid page tables are stored in memory (others on disk)



PFN	V	P
NULL	I	N
PFN	V	N
NULL	I	N
PFN	V	P

Level 1 page table

invalid, no L2 page table

NULL	I	N
NULL	I	N
PFN	V	N
NULL	I	N
PFN	V	P

Level 2 page table

valid and present L2 page tables are stored in memory

PFN	V	N
NULL	I	N
PFN	V	N
NULL	I	N
PFN	V	N

Level 2 page table

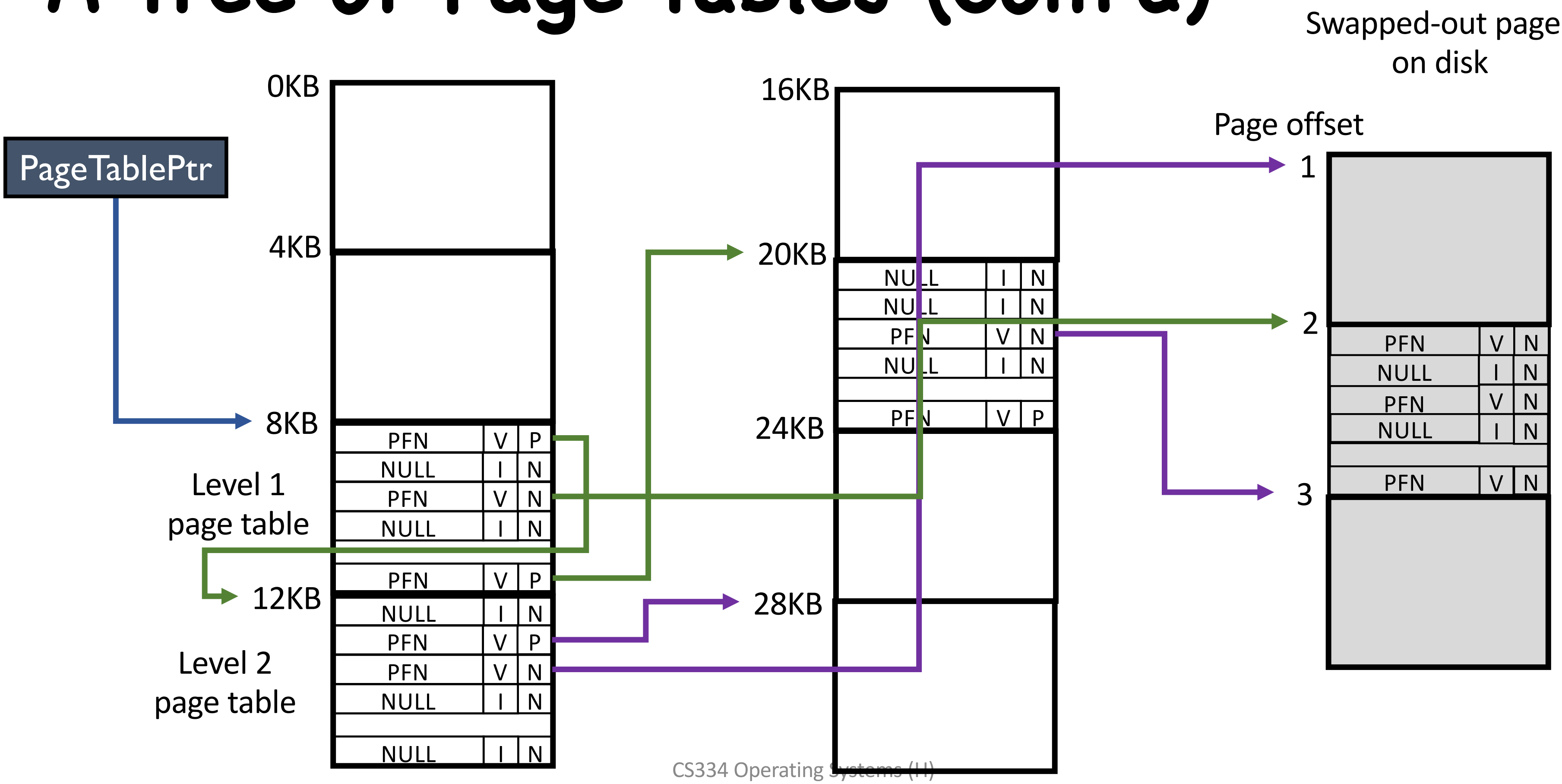
valid but non-present L2 page tables are stored on disk

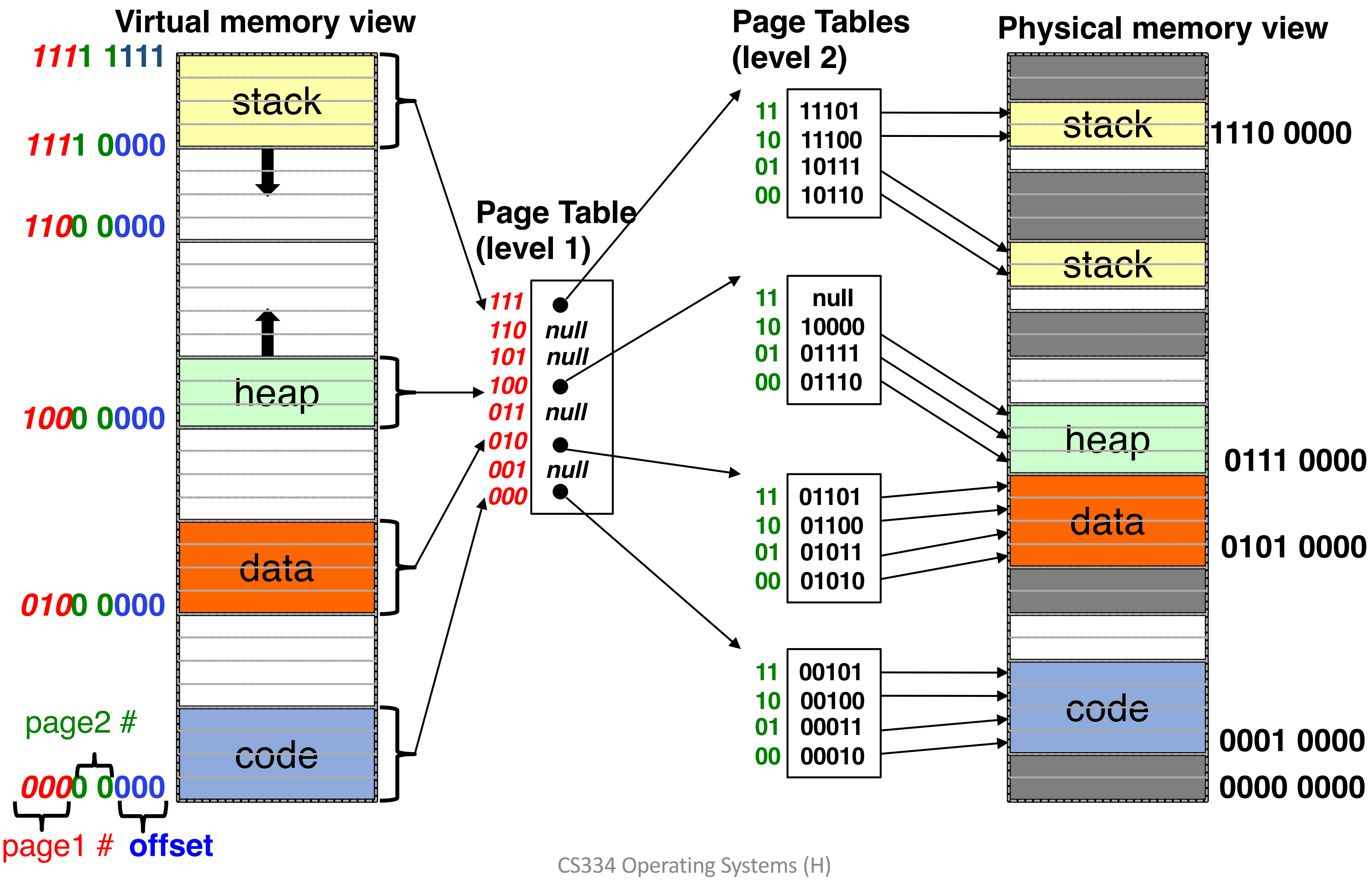
……

NULL	I	N
PFN	V	P
PFN	V	N
NULL	I	N
NULL	I	N

Level 2 page table

A Tree of Page Tables (Cont’d)





Virtual memory view

1111 1111

1111 0000

1100 0000

1000 0000

0100 0000

page2 #

0000 0000

page1 # offset

stack

heap

data

code

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

Physical memory view

stack

stack

heap

data

code

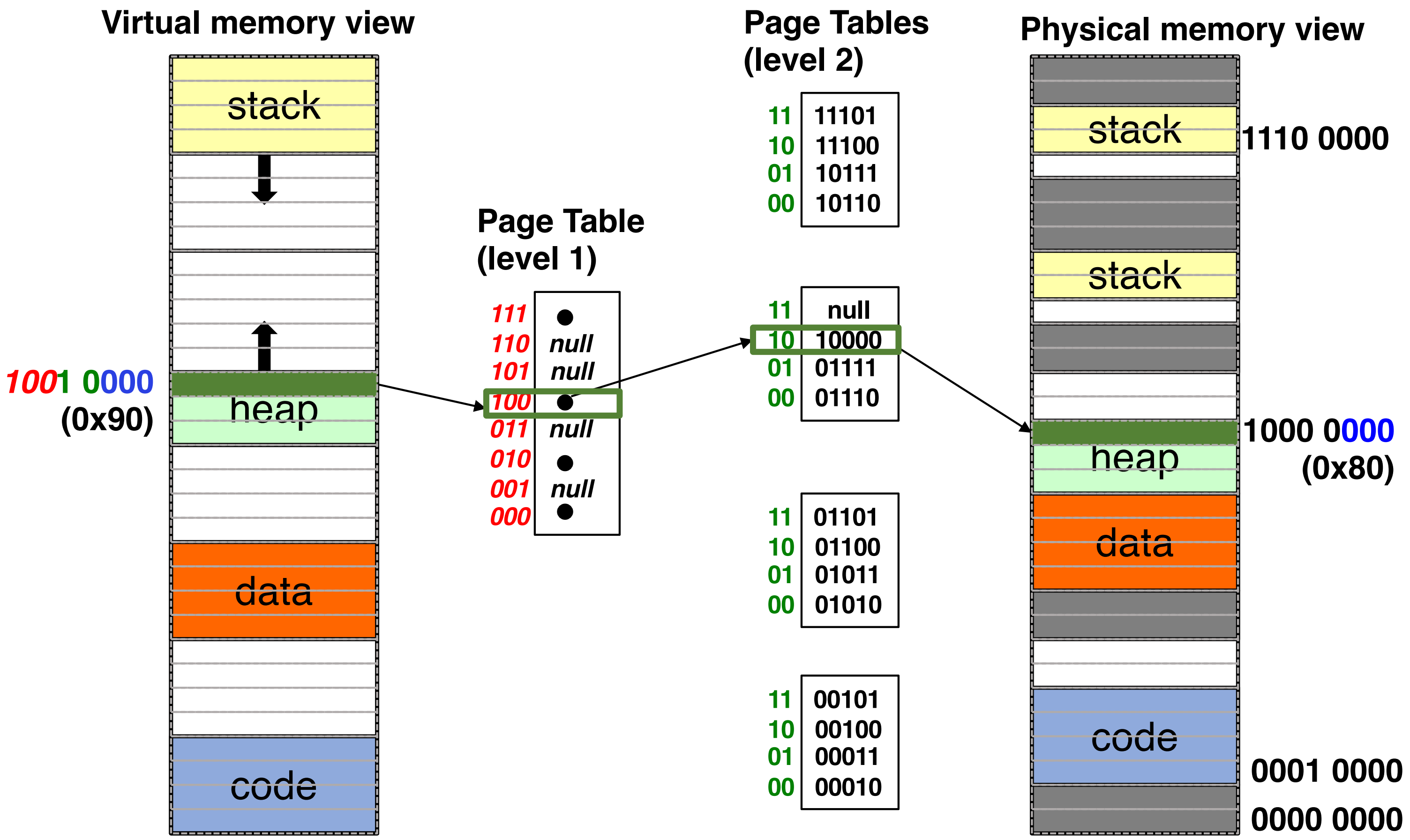
1110 0000

0111 0000

0101 0000

0001 0000

0000 0000



Virtual memory view

stack

heap

data

code

1001 0000

(0x90)

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

Physical memory view

stack

stack

heap

data

code

1110 0000

1000 0000

(0x80)

0001 0000

0000 0000

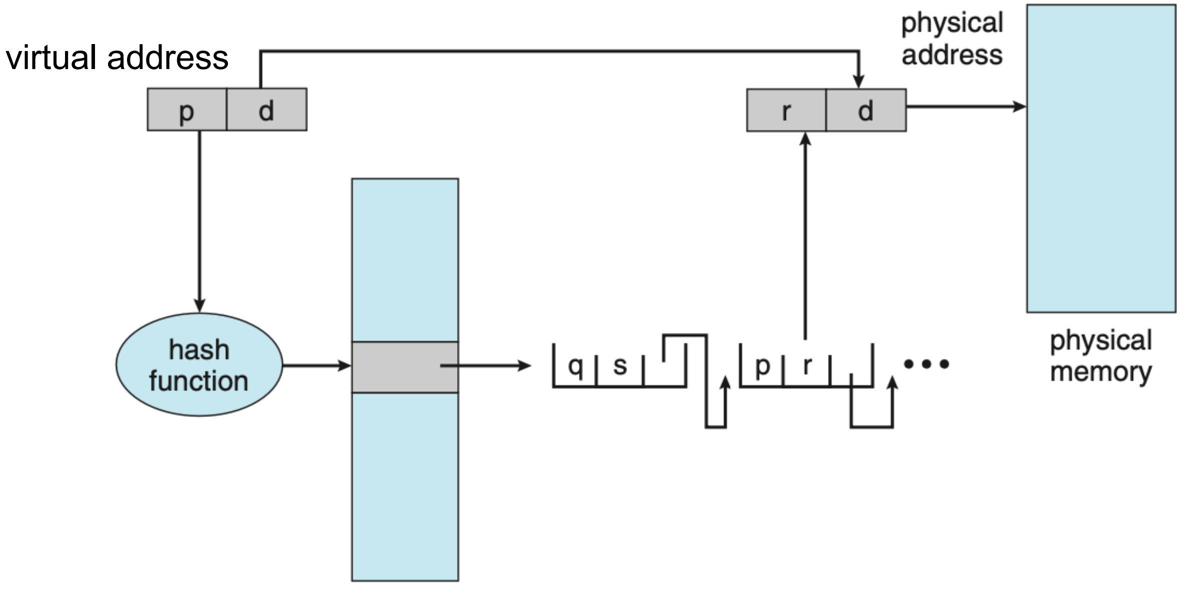
Other Page Table Structures

Page Table Structures

- Hierarchical page tables
  - 2-level page tables
  - 3-level page tables
  - 4-level page tables
- Hashed Page Tables
- Inverted Page Tables

Hashed Page Tables

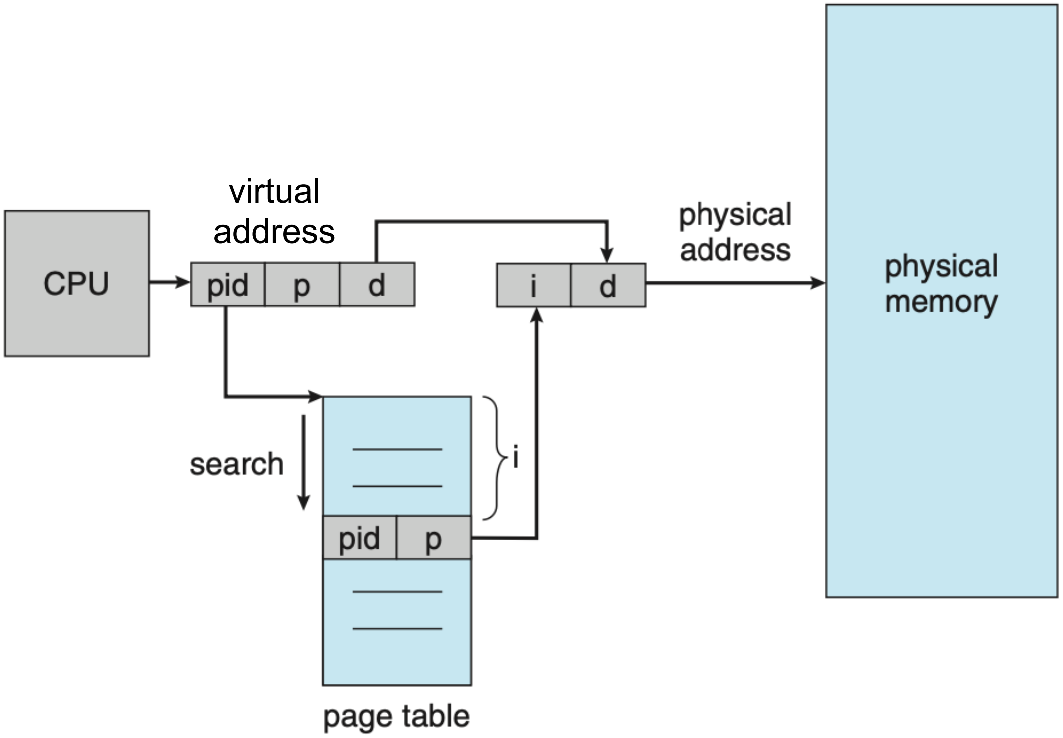
- Hash function
  - input: VPN
  - output: index in the hashed page table
- Collision handling
  - A linked list
  - Each element consists of three fields: (1) VPN, (2) PFN, and (3) pointer to the next element



hashed page table

Inverted Page Tables

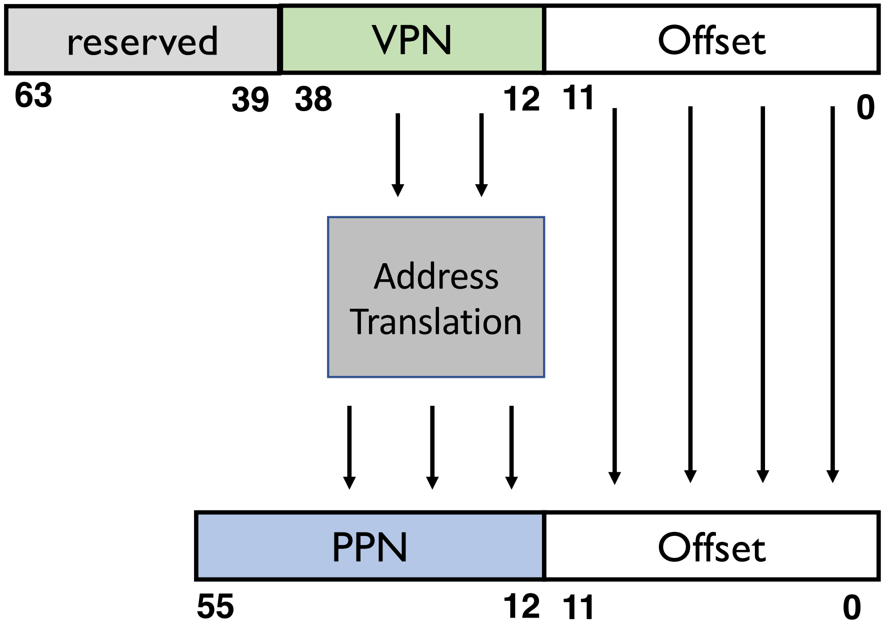
- One page table for the whole system
  - Used in 64-bit UltraSPARC and PowerPC
- Each entry corresponds to one physical page frame
  - Process ID and VPN
- Page table lookup requires linear search of the entire table
- Memory sharing is hard



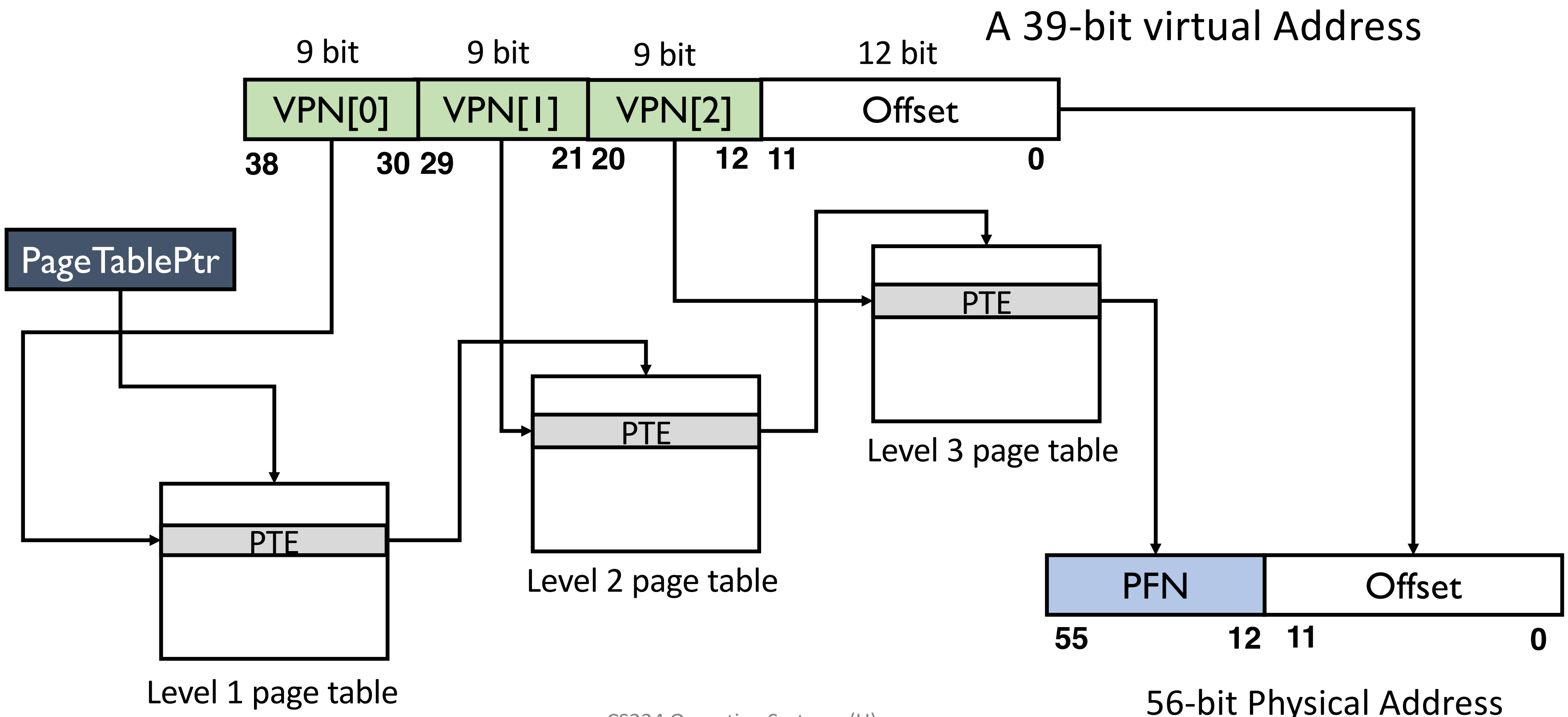
Real-world Paging Schemes

Virtual Memory on RISC-V

- RISC-V supports multiple MMU
  - For RV32: SV32
  - For RV64: SV39 and SV48
- Here we introduce SV39
  - Page size: 4KB
  - Virtual address: 39 bits
    - remaining bits of total 64 bits reserved
  - Physical address: 56 bits

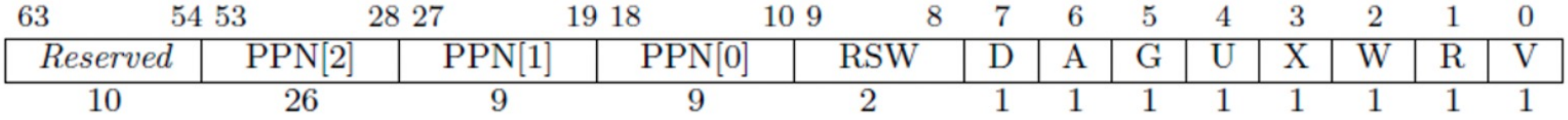


SV39: Three Levels of Page Tables



SV39: Page Table Entries

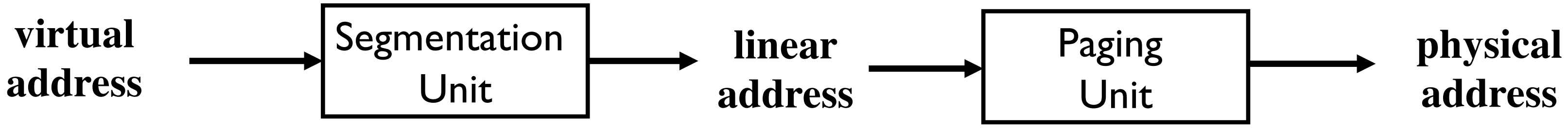
- A PTE in SV39 takes 8 bytes (64 bits)
  - Bit 8-0 status bits
  - Bit 53-10 is PFN (or physical page number, PPN)
  - Bits 63-54 reserved
- The 9 status bits
  - D (dirty), A (accessed), V (valid)
  - G (Global): G=1 the page is mapped in all address spaces
  - U (User): U-mode code may access this page
  - RSW: reserved for s-mode



63 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

Virtual Memory on IA-32

- Intel's 32-bit CPU (IA-32) uses two stage address translation: segmentation + paging
- A virtual address contains a 16-bit segment selector and 32-bit offset



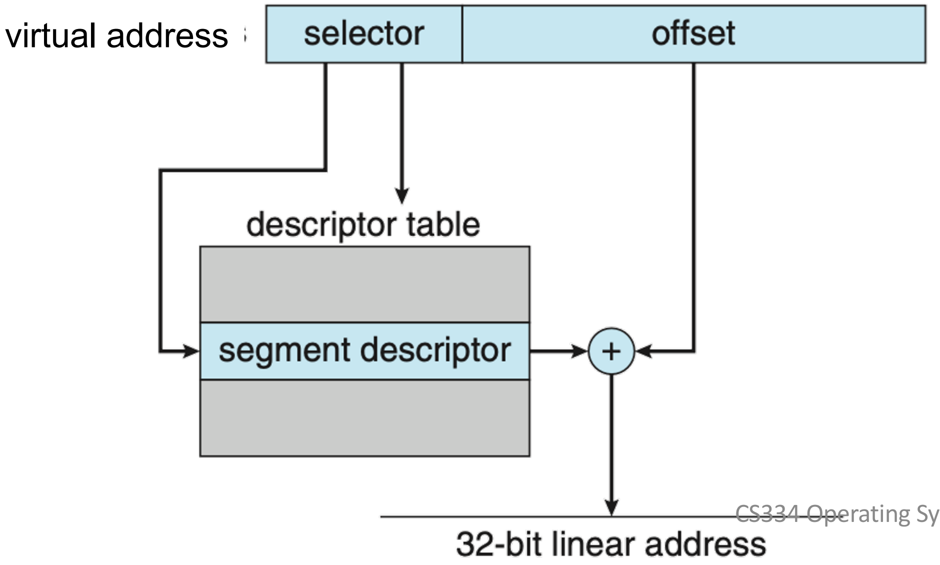
virtual address → Segmentation Unit → linear address → Paging Unit → physical address

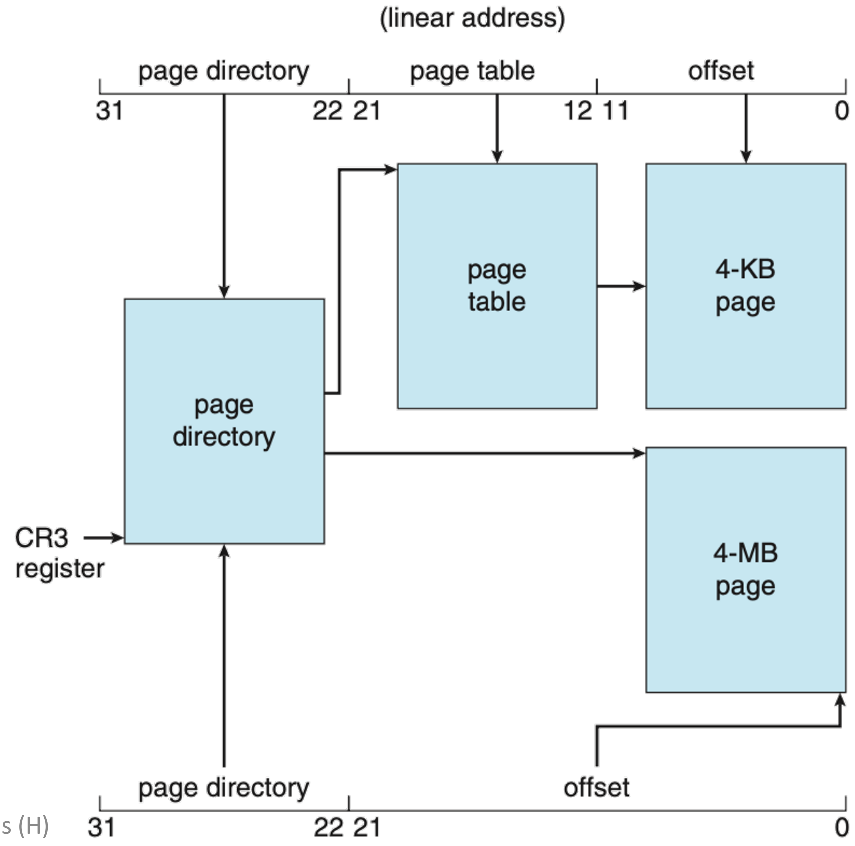
virtual address

16 bit	32 bit
selector	Offset

Virtual Memory on IA-32

- Two descriptor tables: GDT & LDT
- Six segment register to cache segment base addresses



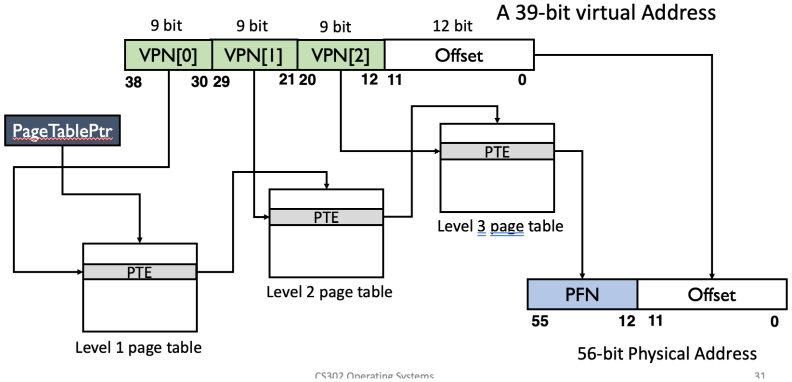


Translation Lookaside Buffer

Issues of Paging

- Time complexity
  - Extra memory references during address translation
  - Three-level page tables requires 3 additional memory reads
    - If every memory reference needs 4 memory reads

Question:
- How to speed up address translation and avoid extra memory reads?



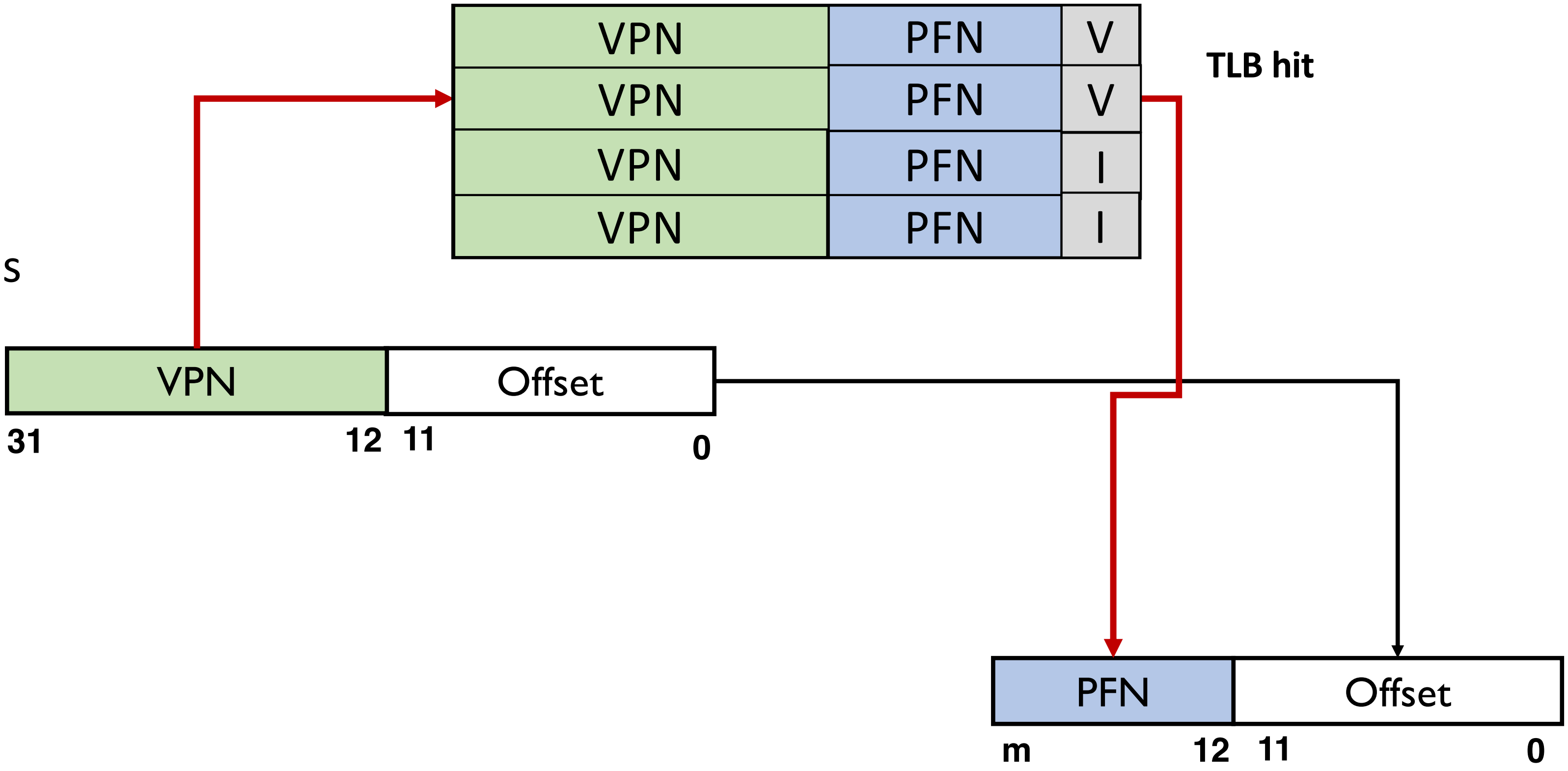
Translation Lookaside Buffer

- A translation lookaside buffer (TLB) is a hardware cache that is part of the MMU
  - A cache for the PTEs: holding a translation likely to be re-used
    - Replacement policy: LRU, FIFO, random
  - Each entry holds mapping of a virtual address to a physical address

- Before a virtual-to-physical address translation is to be performed, TLB is looked up using VPN
  - TLB hit: VPN is found, and the PFN of the same entry used
  - TLB miss: VPN not found, page table walk

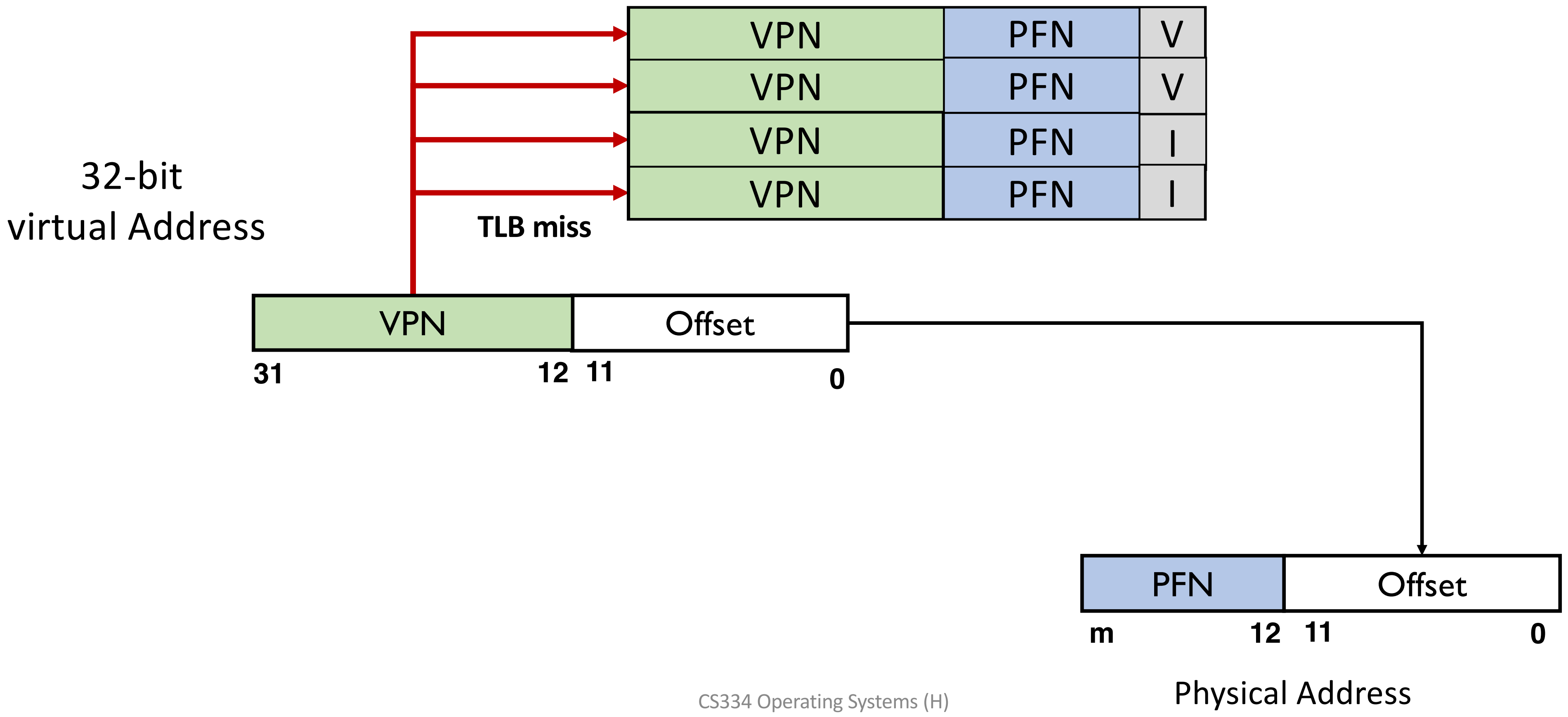
Translation Lookaside Buffer (Cont'd)

32-bit
virtual Address

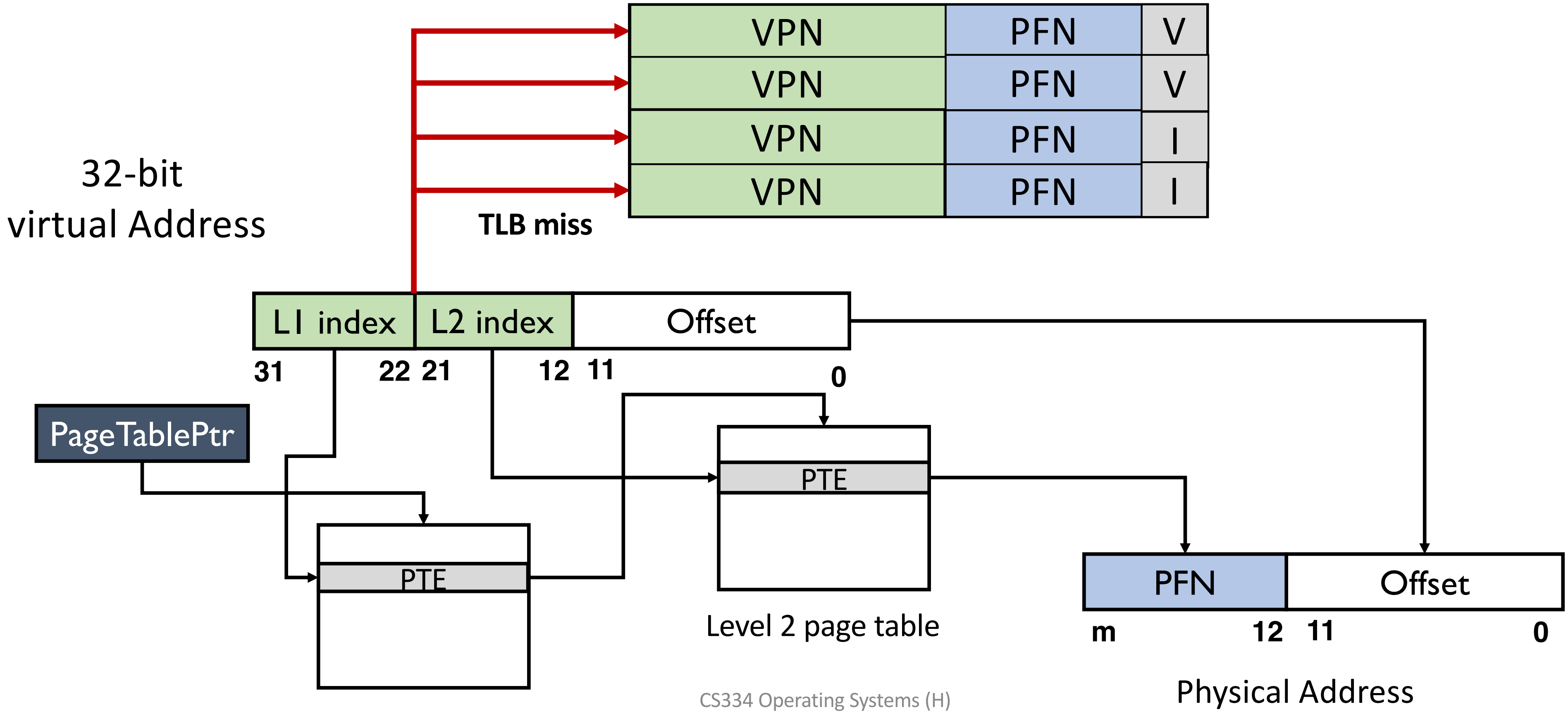


Physical Address

Translation Lookaside Buffer (Cont'd)

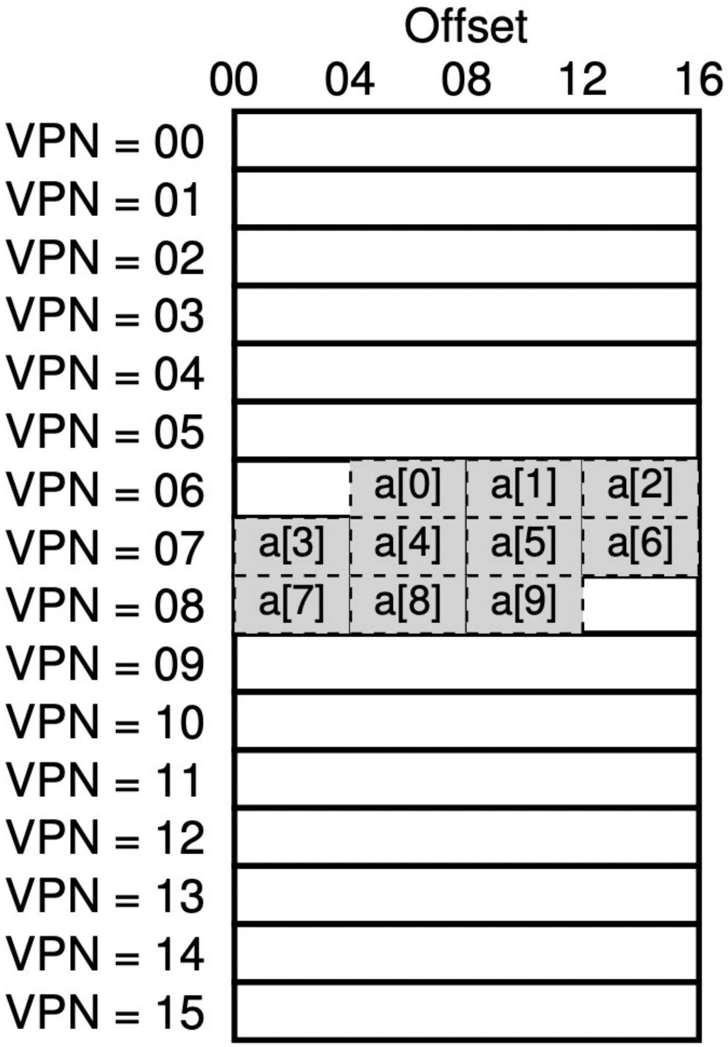


Translation Lookaside Buffer (Cont'd)



Why Does TLB Work?

- Ideally one page table walk for the entire page
  - TLB is smaller than page table, but
- Spatial locality
  - Sequentially executed instructions, local variables (on stack), arrays (on heap) likely on the same page
  - E.g., accessing a[0] to a[9]
    - m,h,h,m,h,h,h,m,h,h -> 70% hit rate
    - Large page size increases hit rate
- Temporal locality
  - Accesses to the same page tend to be close in time



Issues with Context Switch

- Two process may use the same virtual address
  - P1: 100 -> 110
  - P2: 100 -> 170
- Solutions
  - Flush TLB upon context switch
    - Invalidate all entries: V->I
  - Extending TLB with address space ID
    - No need to flush tlb

VPN	PFN	valid
-	-	I
100	110	V
-	-	I
100	170	V

VPN	PFN	valid	ASID
-	-	I	-
100	110	V	1
-	-	I	-
100	170	V	2

Thank you!