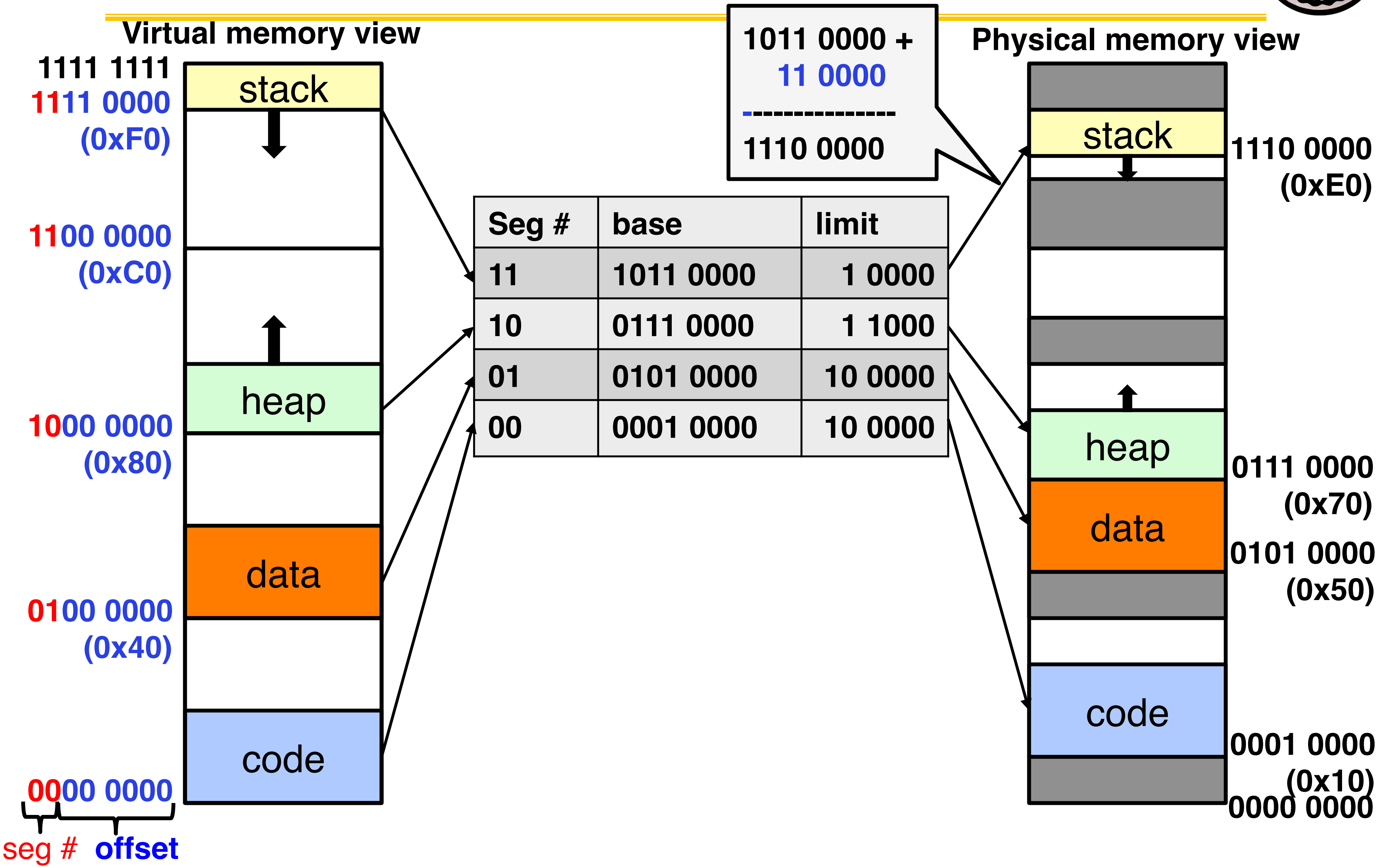
# CS162 - Operating Systems and Systems Programming

## Address Translation - Continued

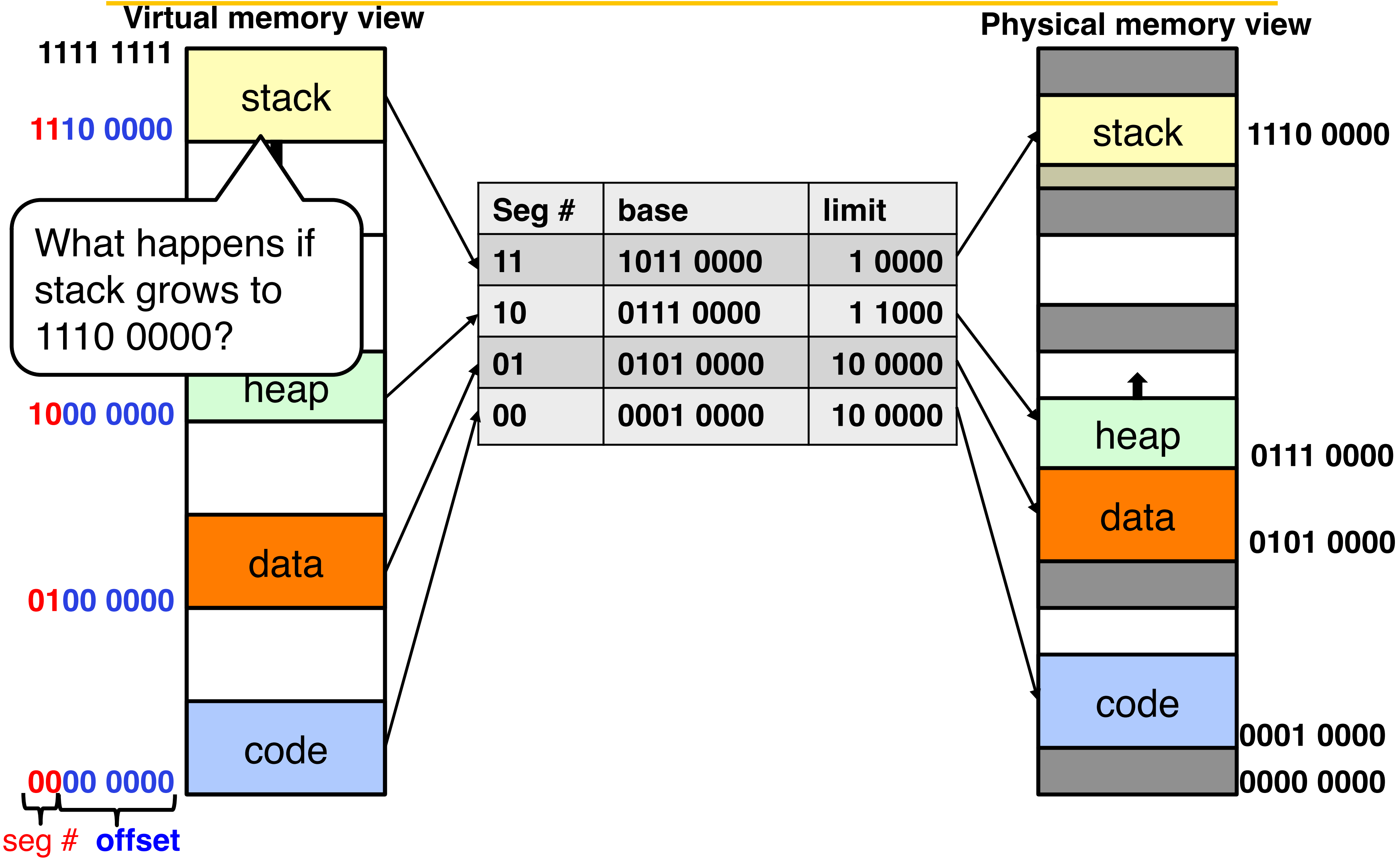
David E. Culler
http://cs162.eecs.berkeley.edu/
Lecture #15
Oct 1, 2014

# Summary: Address Segmentation



| Seg # | base | limit |
|---|---|---|
| 11 | 1011 0000 | 1 0000 |
| 10 | 0111 0000 | 1 1000 |
| 01 | 0101 0000 | 10 0000 |
| 00 | 0001 0000 | 10 0000 |

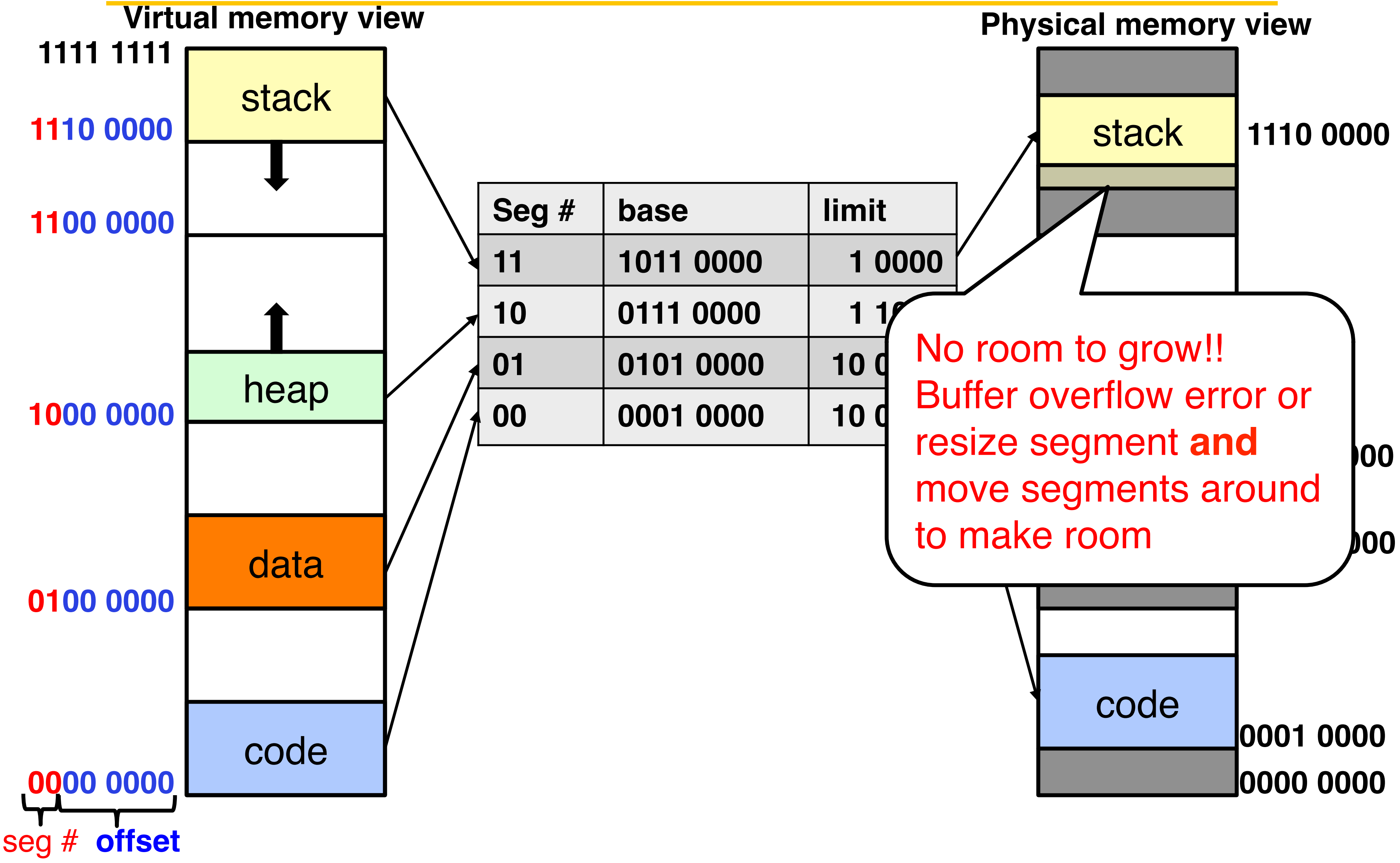
# Summary: Address Segmentation



| Seg # | base | limit |
|---|---|---|
| 11 | 1011 0000 | 1 0000 |
| 10 | 0111 0000 | 1 1000 |
| 01 | 0101 0000 | 10 0000 |
| 00 | 0001 0000 | 10 0000 |

What happens if stack grows to 1110 0000?

# Recap: Address Segmentation



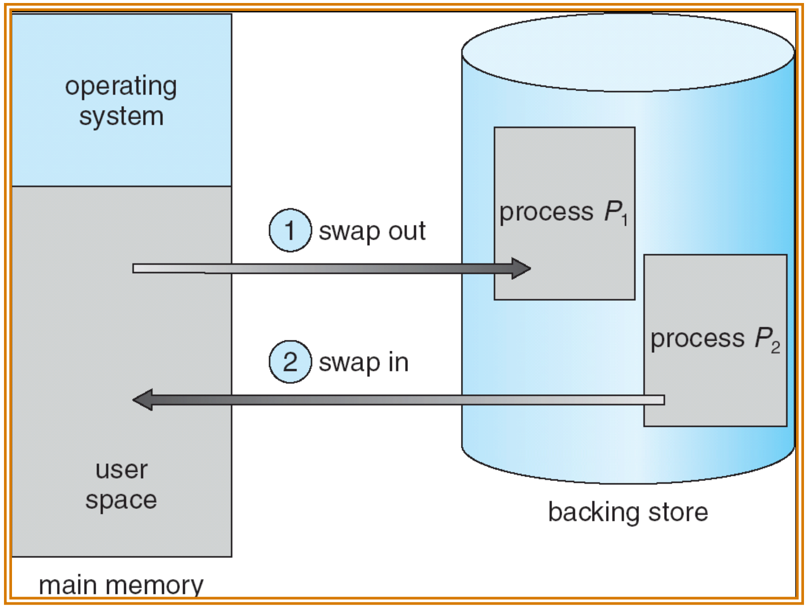
| Seg # | base | limit |
|---|---|---|
| 11 | 1011 0000 | 1 0000 |
| 10 | 0111 0000 | 1 10 |
| 01 | 0101 0000 | 10 0 |
| 00 | 0001 0000 | 10 0 |

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

# How do we run more programs than fit in memory ?

# 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

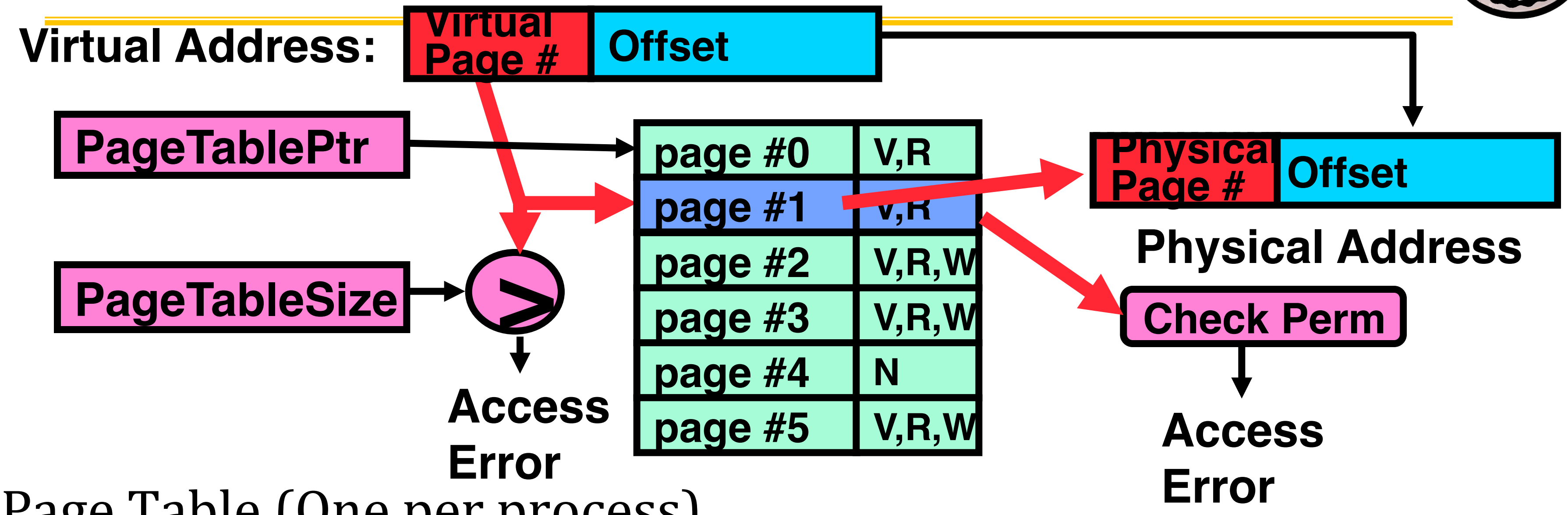
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks

# 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?



Virtual Address: **Virtual Page #** | **Offset**

**PageTablePtr**

**PageTableSize**

>

Access Error

| | |
|---|---|
| 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 |

**Physical Page #** | **Offset**

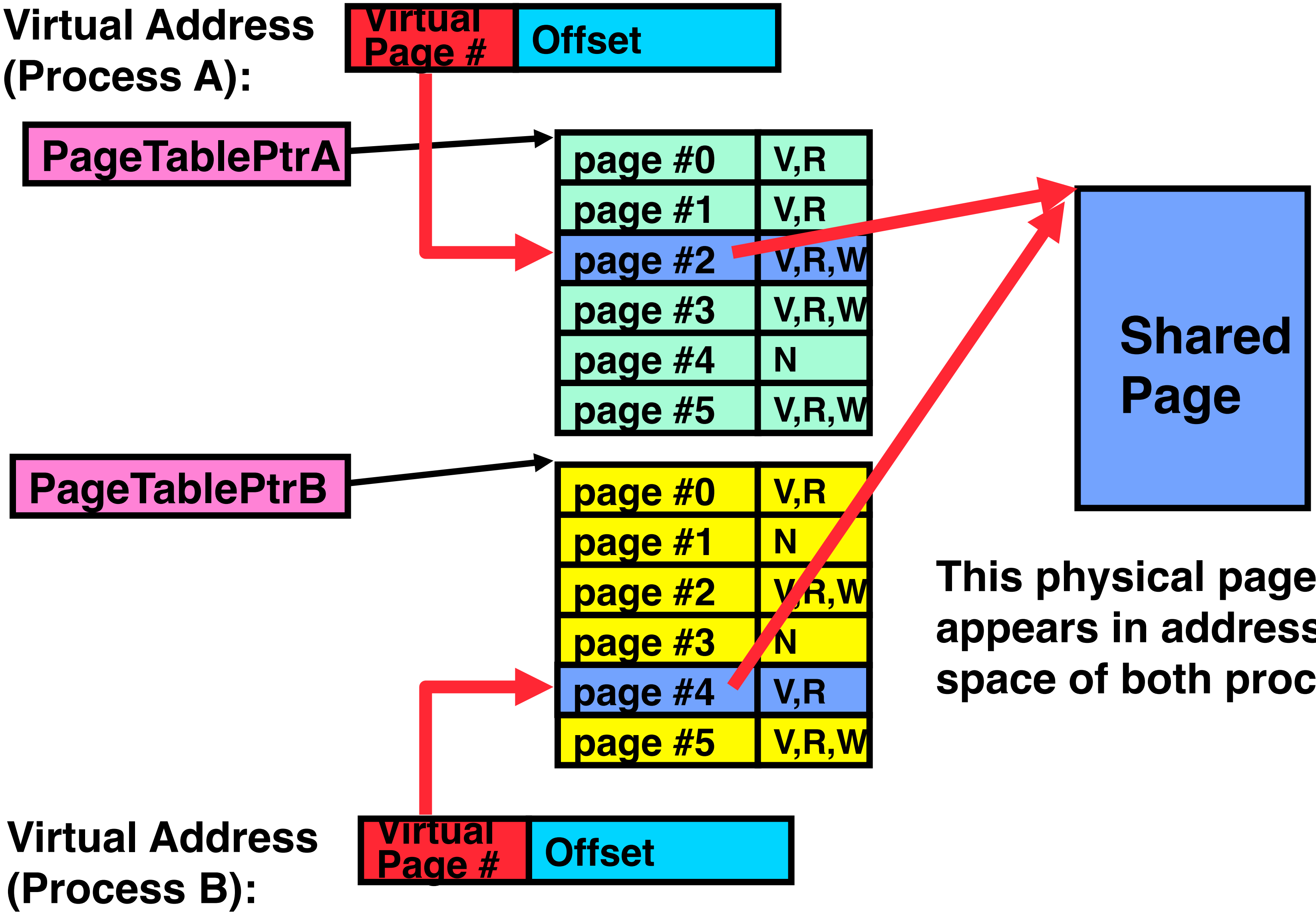
Physical Address

**Check Perm**

Access Error

- Page Table (One per process)
  - Residesin 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, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

# What about Sharing?



Virtual Address (Process A):

| | |
|---|---|
| Virtual Page # | Offset |

PageTablePtrA

| | |
|---|---|
| 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 |

PageTablePtrB

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

Shared Page

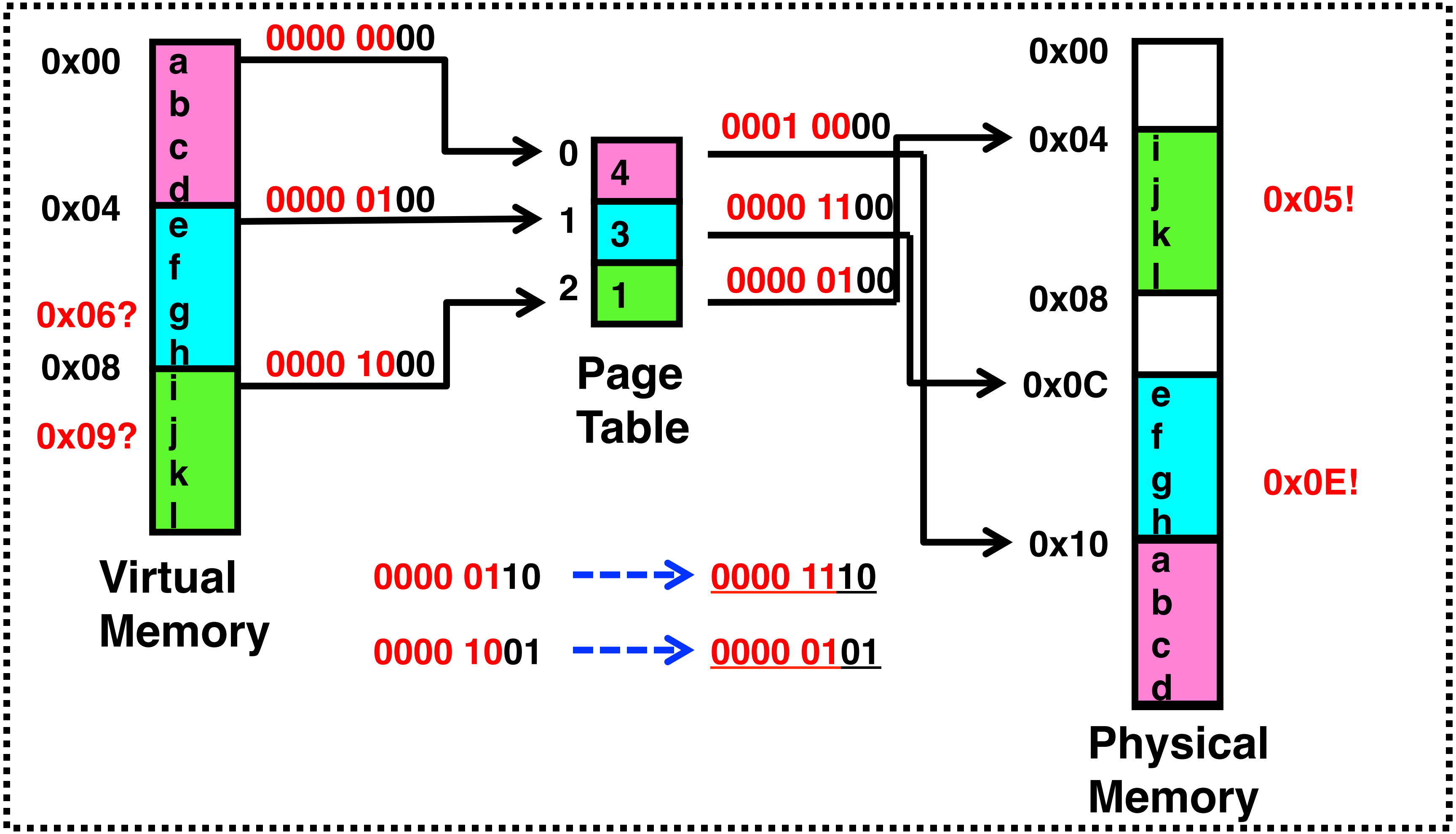
This physical page appears in address space of both processes

Virtual Address (Process B):

| | |
|---|---|
| Virtual Page # | Offset |

# Simple Page Table Example

Example (4 byte pages)

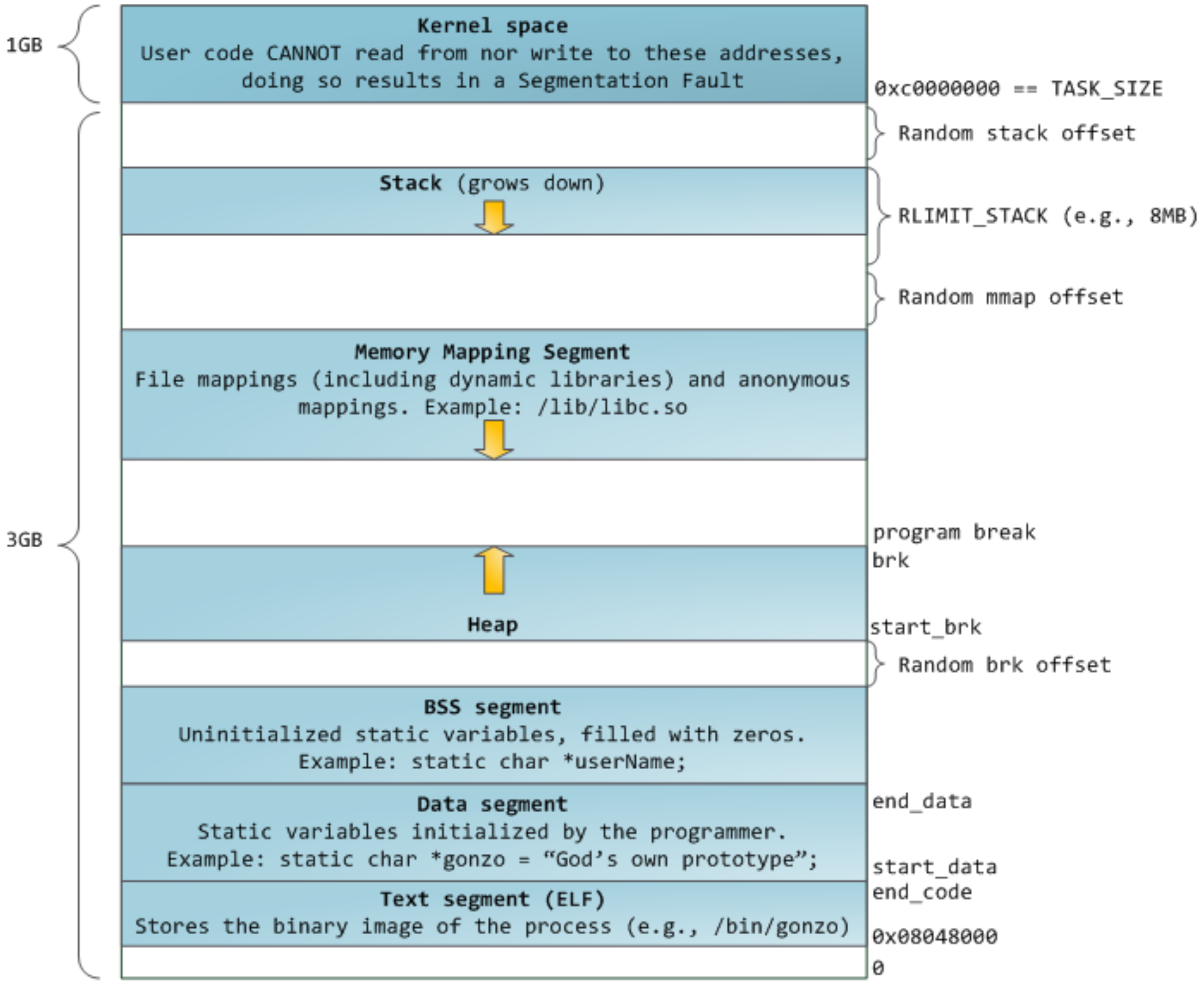


# Page Table Discussion

- 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 really big?
    - Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

# E.g., Linux 32-bit



**Kernel space**
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

**Stack** (grows down)

**Memory Mapping Segment**
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

**Heap**

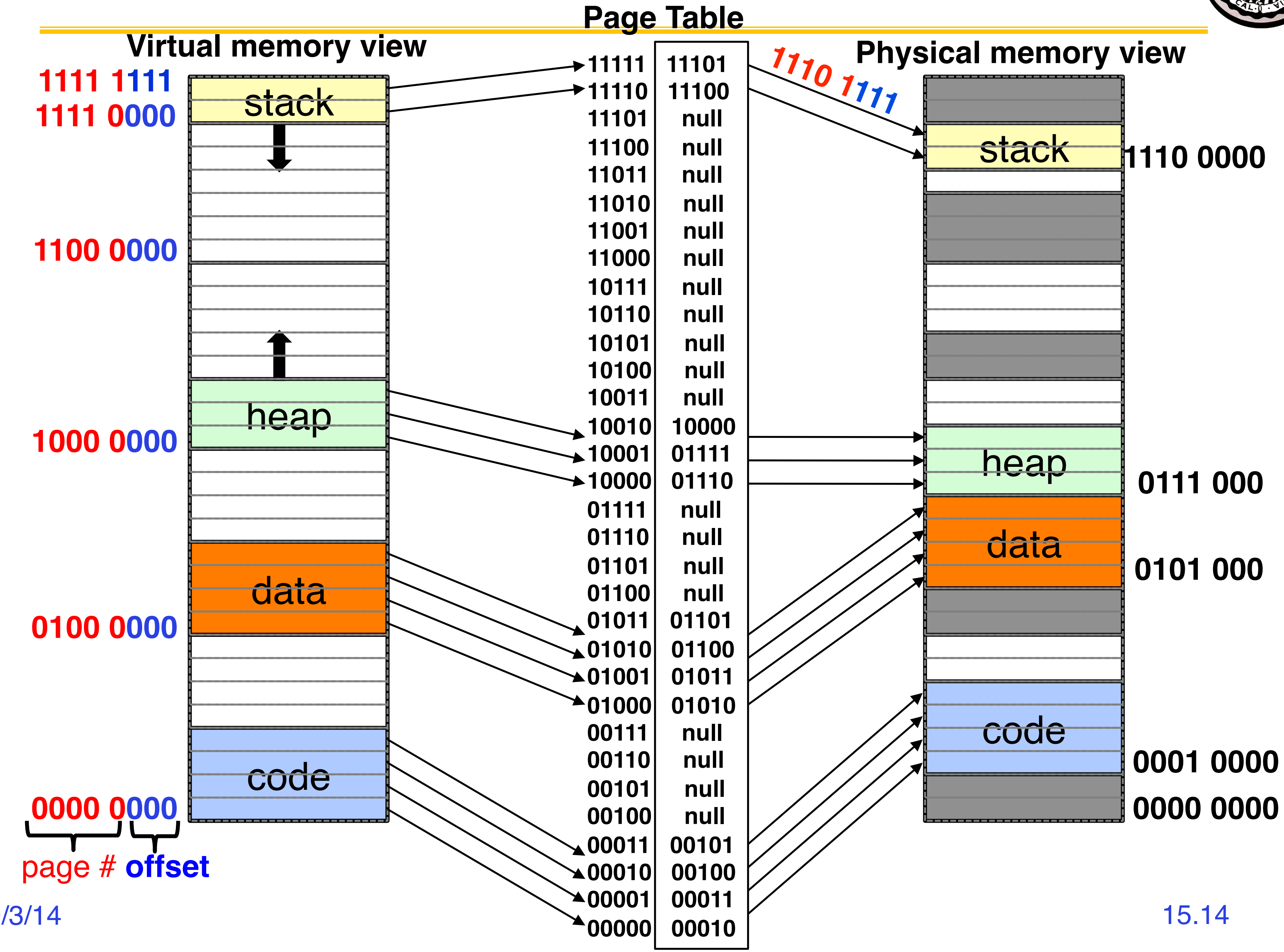
**BSS segment**
Uninitialized static variables, filled with zeros.
Example: static char *userName;

**Data segment**
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";

**Text segment (ELF)**
Stores the binary image of the process (e.g., /bin/gonzo)

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

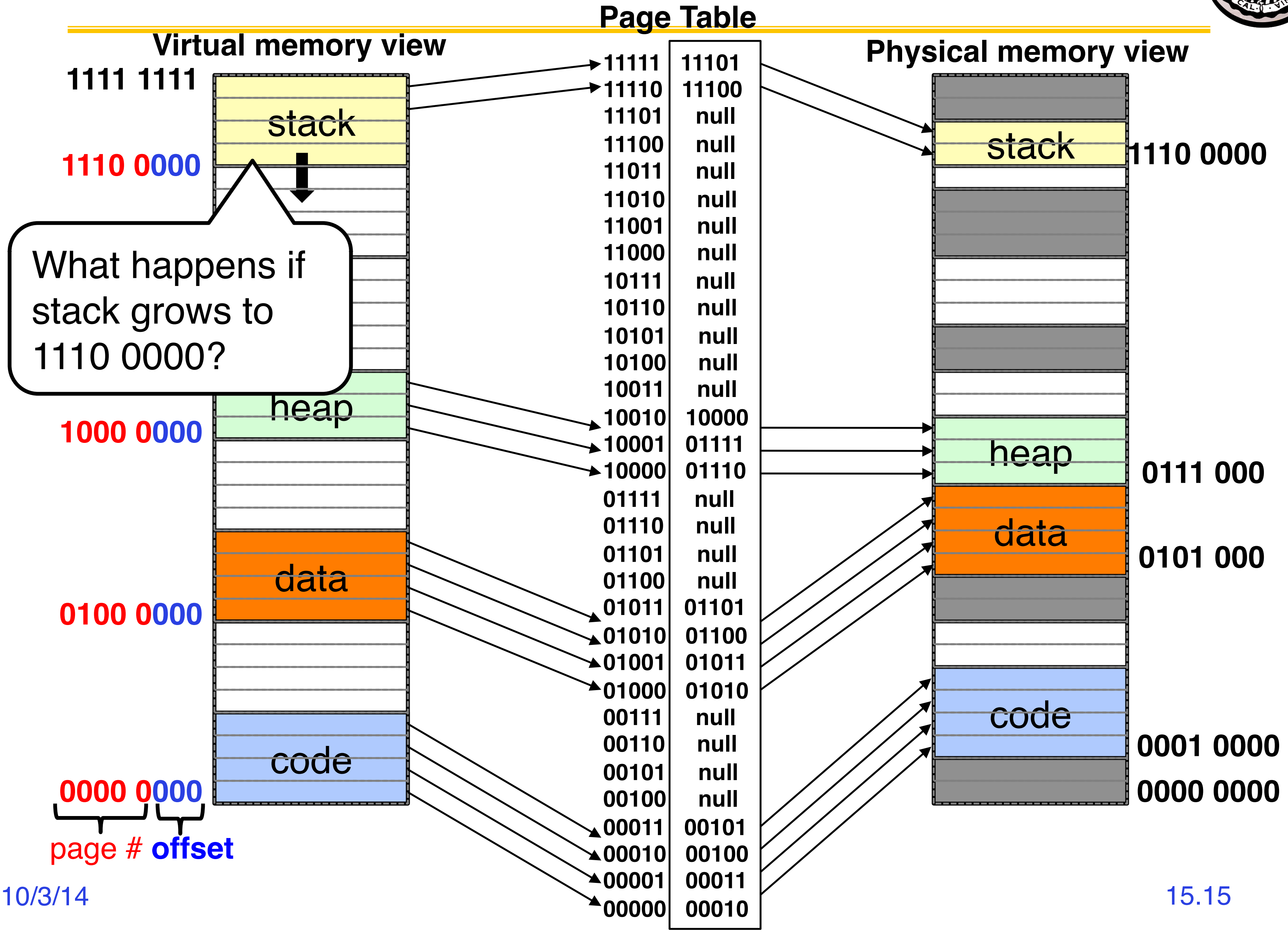
# Summary: Paging



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

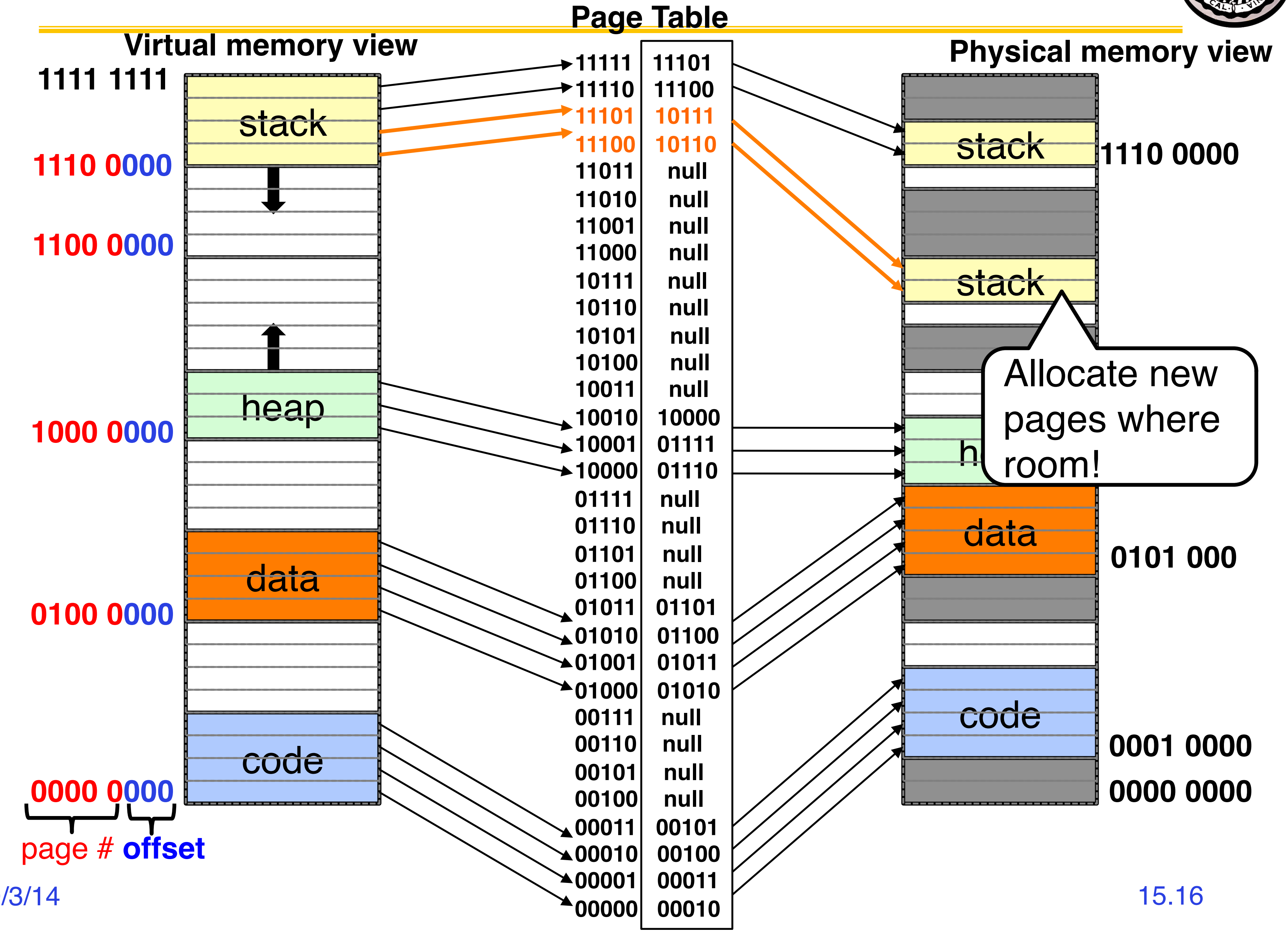


What happens if stack grows to 1110 0000?

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



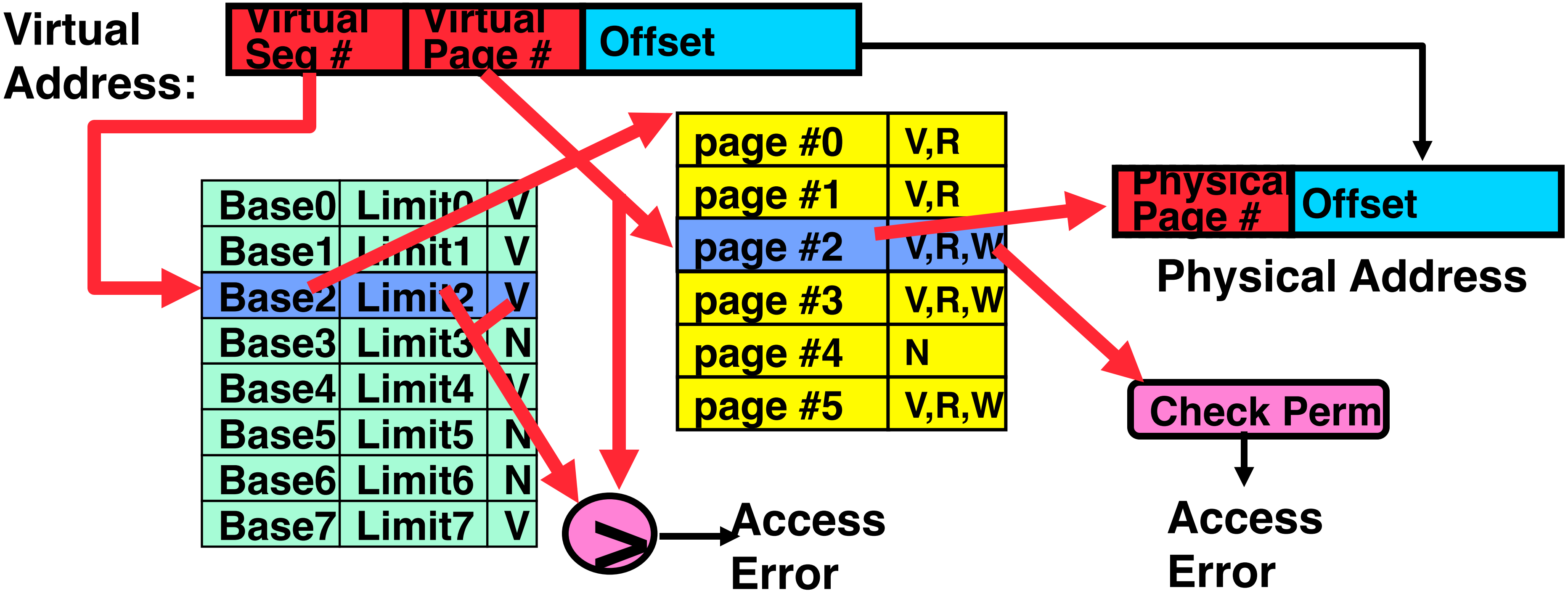
**Page Table**

| | |
|---|---|
| 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 |

Allocate new pages where room!

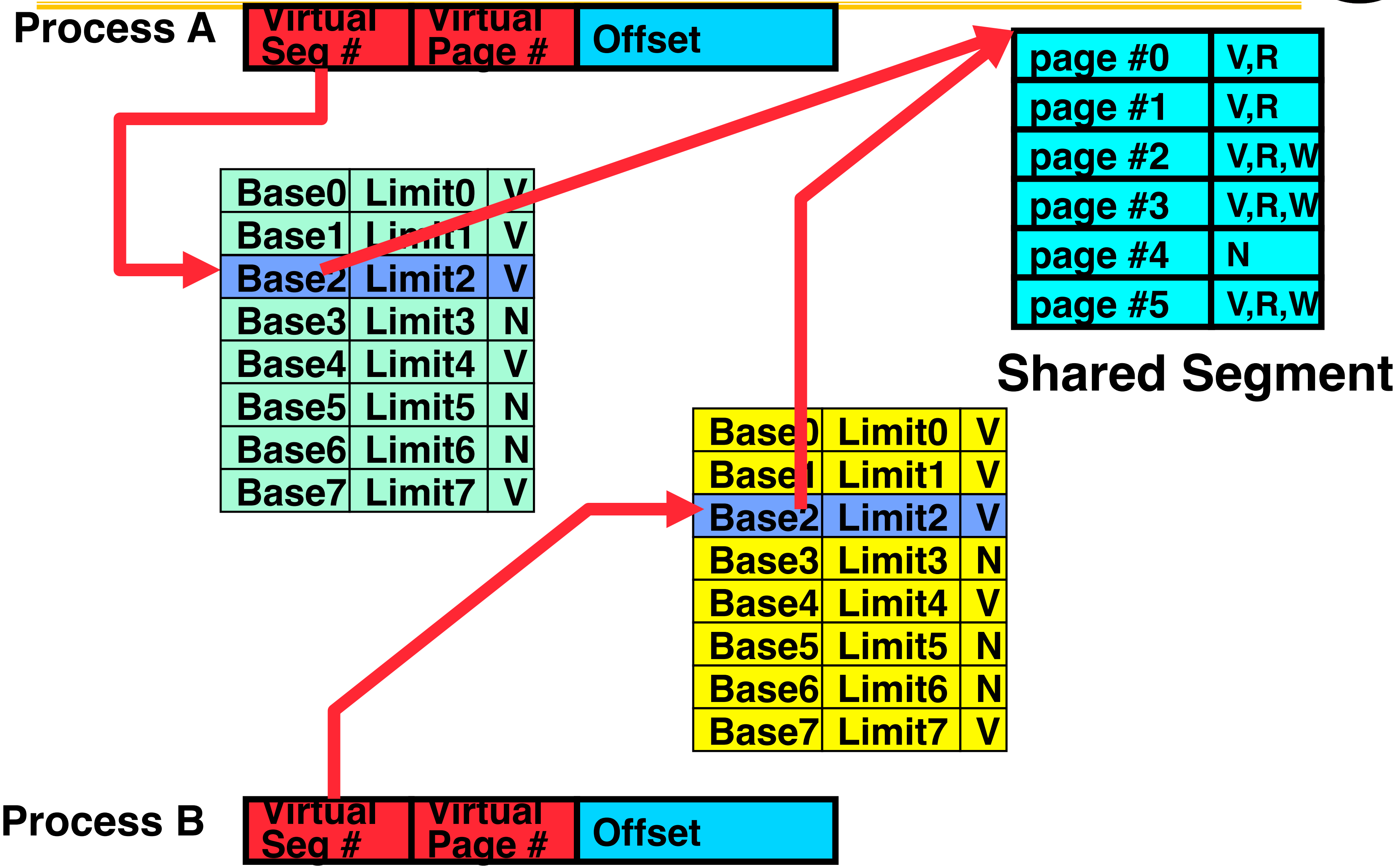
# Multi-level Translation

- 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?
  - 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 |

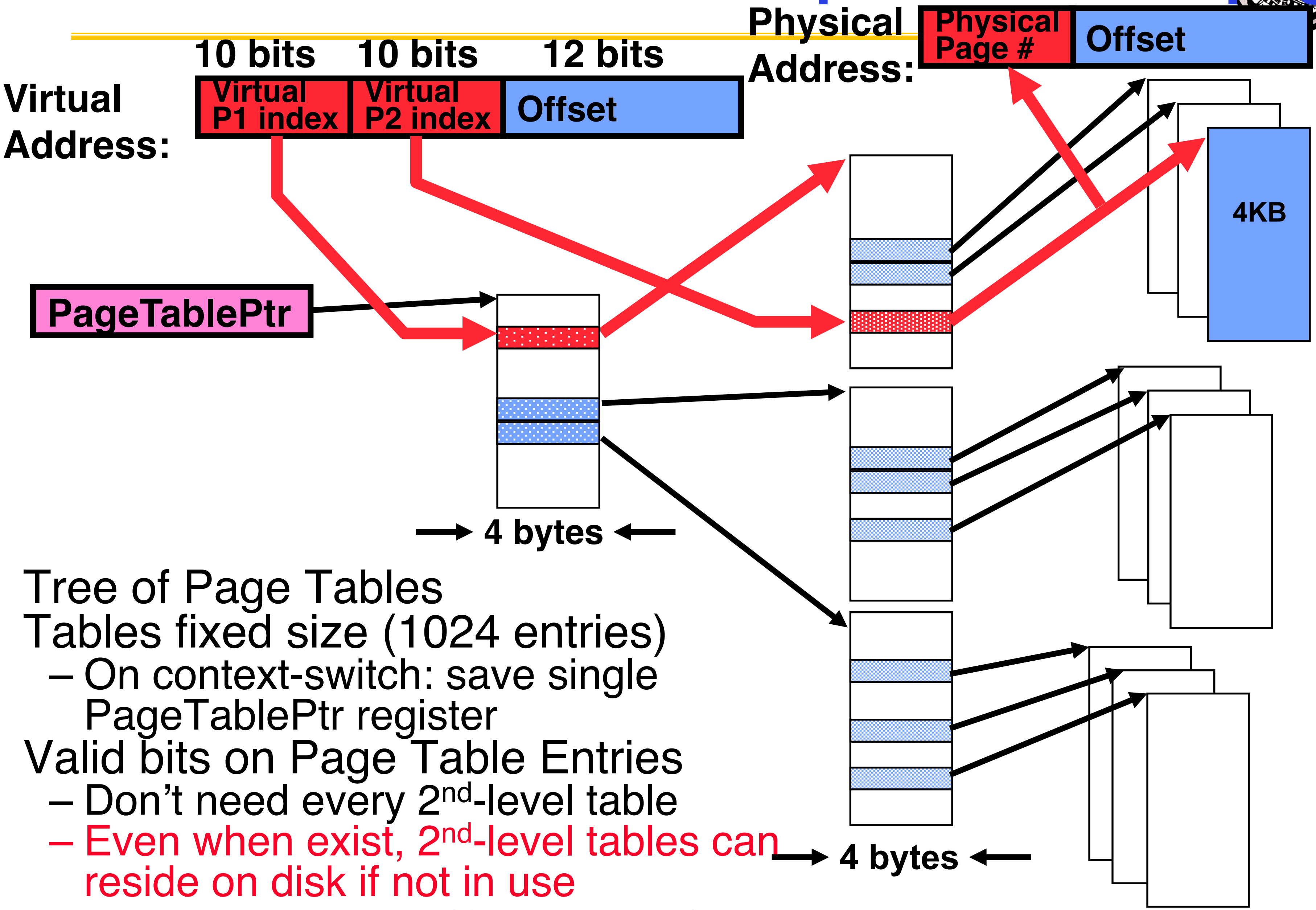
**Shared Segment**

| | | |
|---|---|---|
| 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 |
|---|---|---|

# Another common example: two-level page

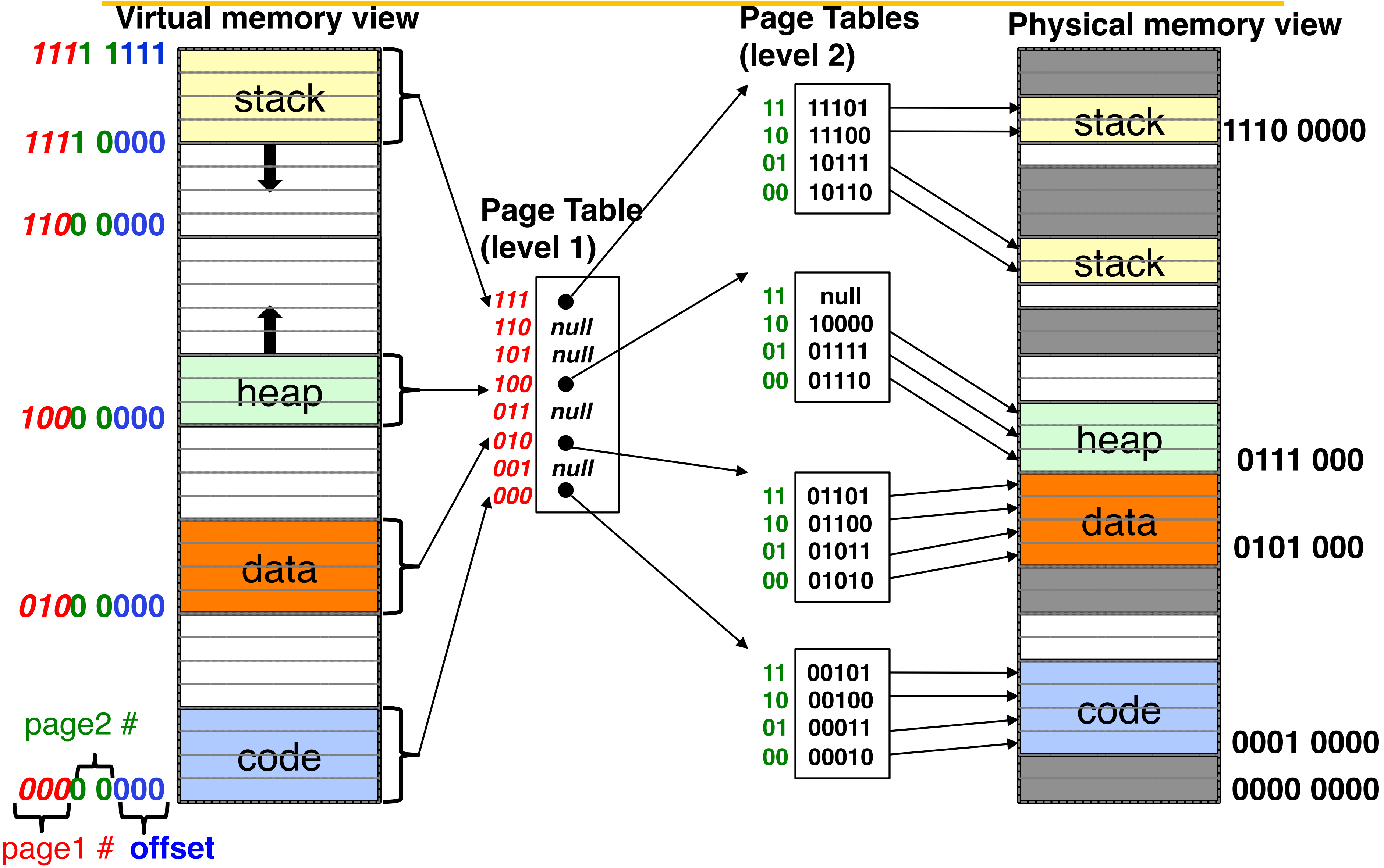


- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

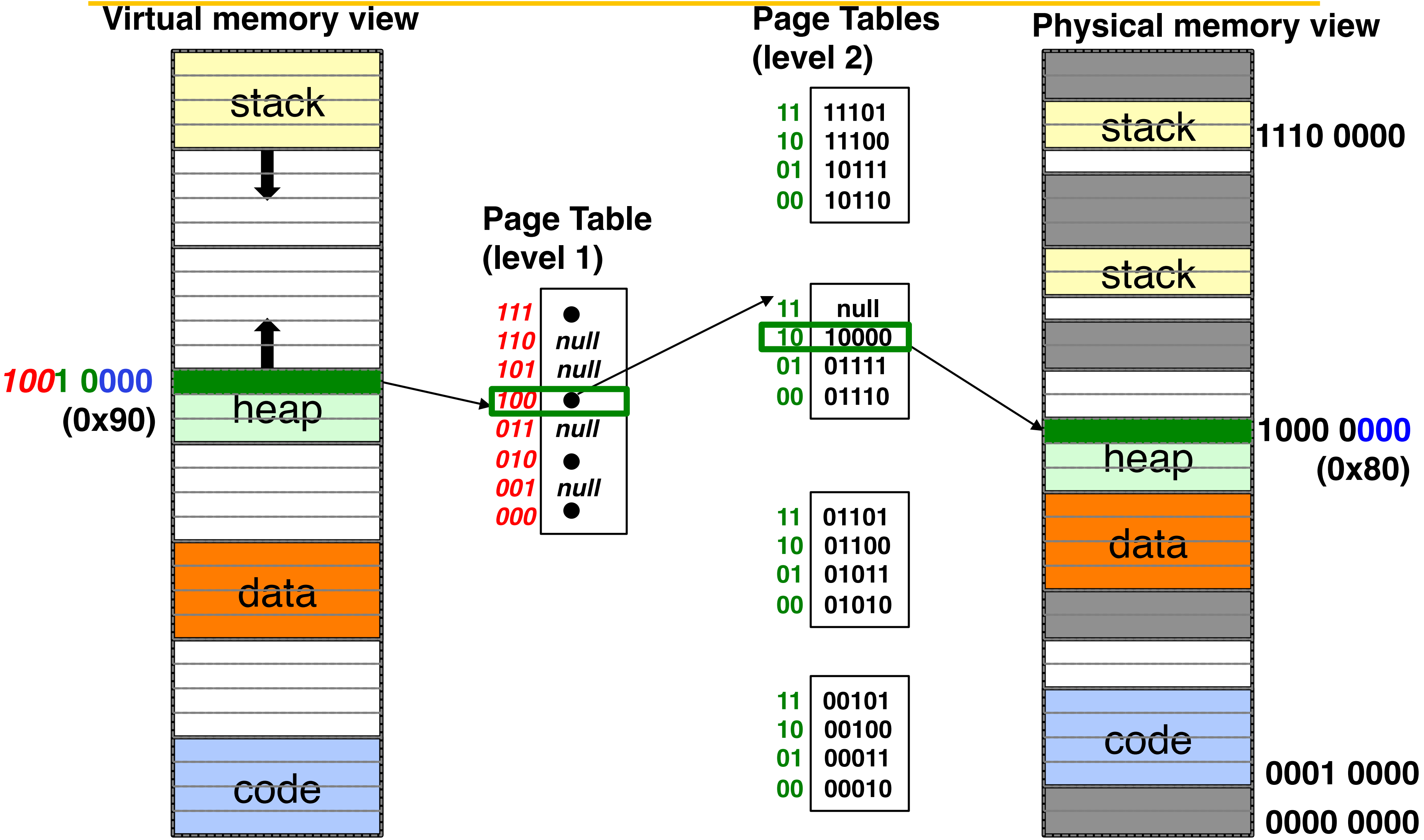
# Multi-level Translation Analysis

- 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
  - Two (or more, if >2 levels) lookups per reference
    - Seems very expensive!

# Summary: Two-Level Paging



# Summary: Two-Level Paging



**Virtual memory view**

stack

1001 0000
(0x90)

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

1110 0000

stack

1000 0000
(0x80)

heap

data

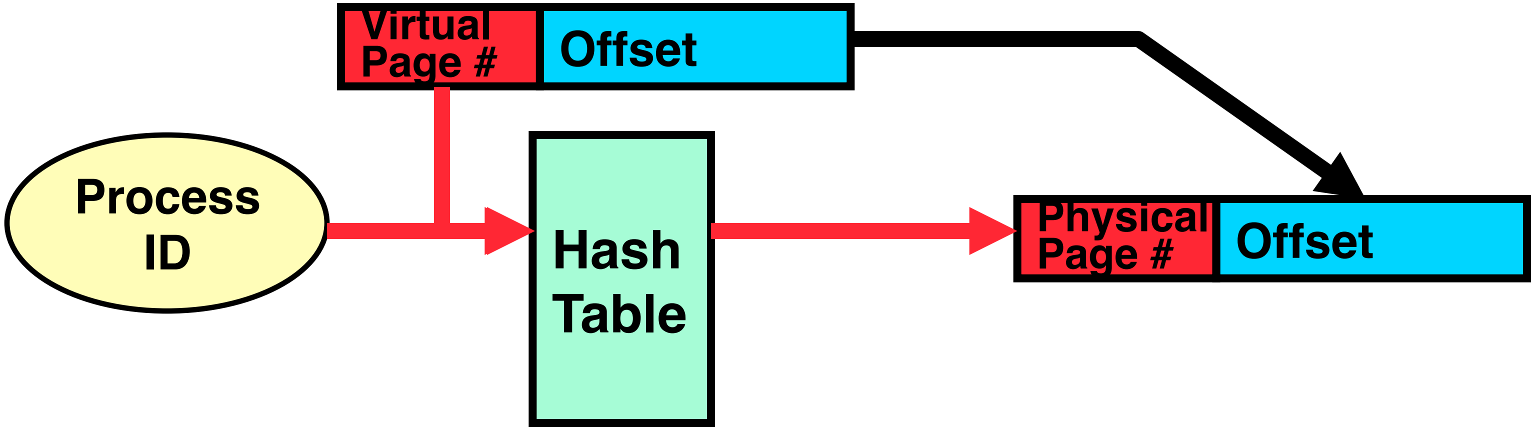
code

0001 0000

0000 0000

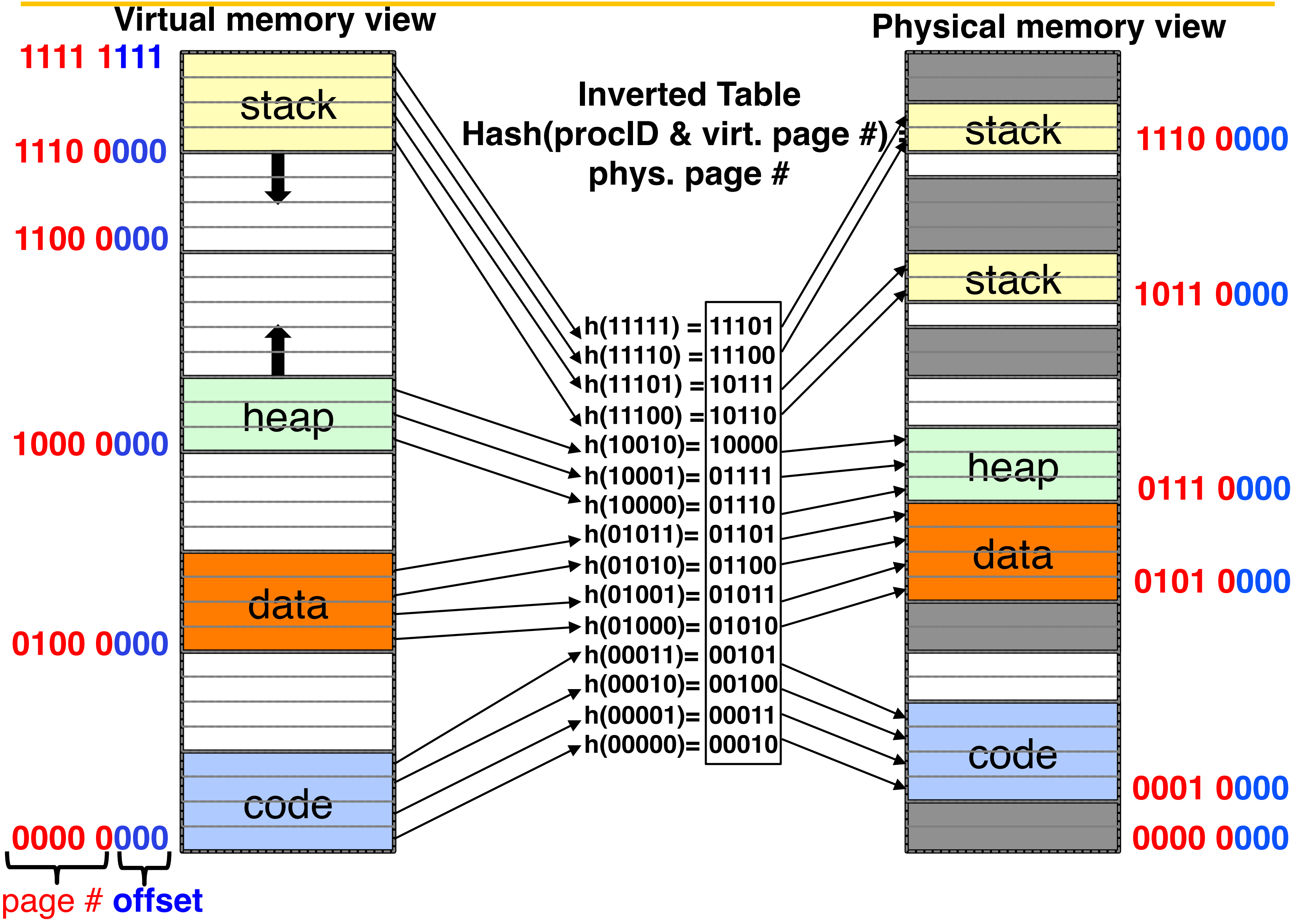
# Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page tables is at least as large as amount of virtual memory allocated to *ALL* processes
  - Physical memory may be much, 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 phy mem (1 entry per phy page)
  - Very attractive option for 64-bit address spaces (IA64, PowerPC, UltraSPARC)
- Cons: Complexity of managing hash chains in hardware

# Summary: Inverted Table



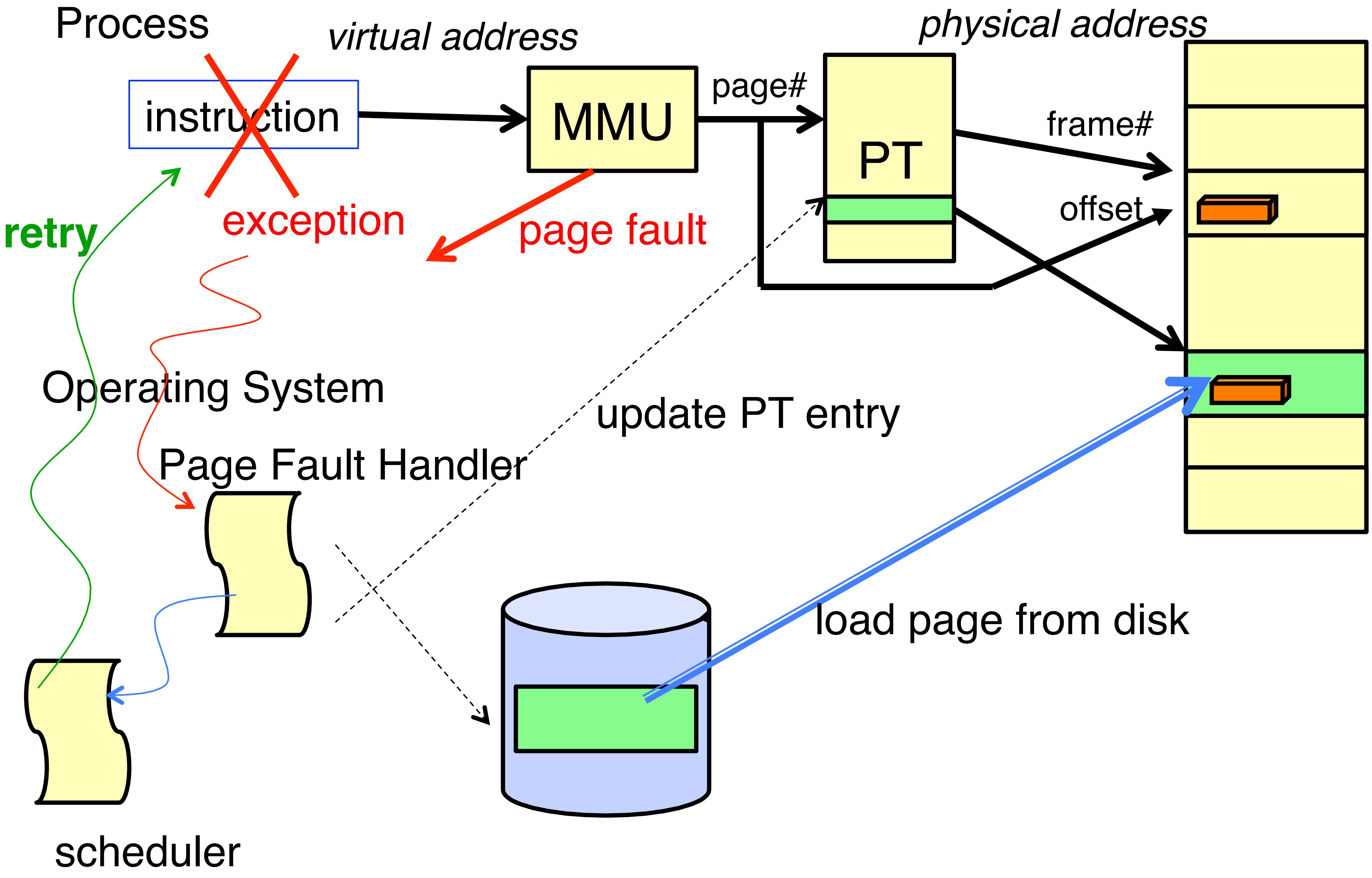
**Inverted Table**
**Hash(procID & virt. page #)**
**phys. page #**

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

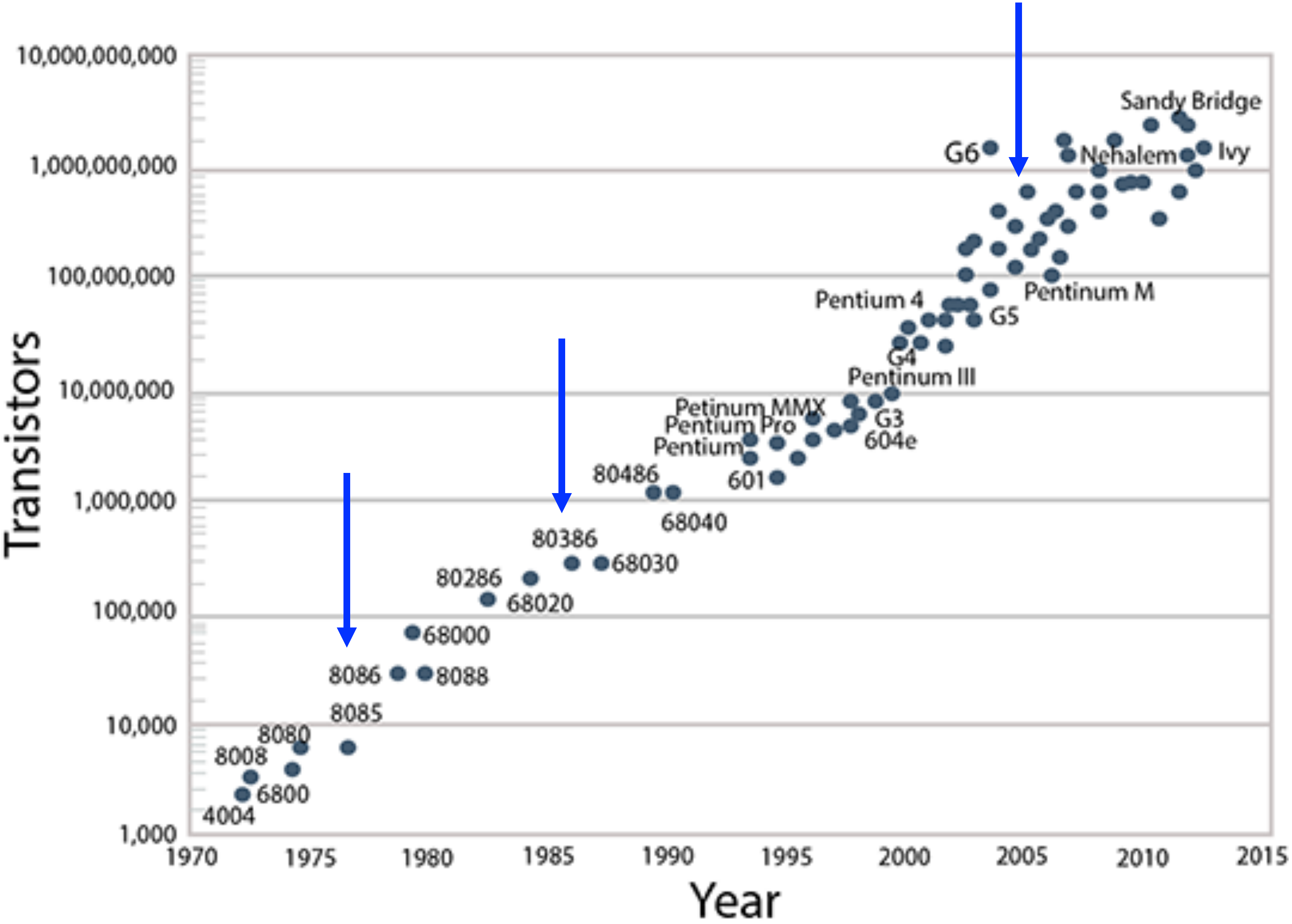
# Address Translation Comparison

| | **Advantages** | **Disadvantages** |
|---|---|---|
| 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<br>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 |

# What happens when …

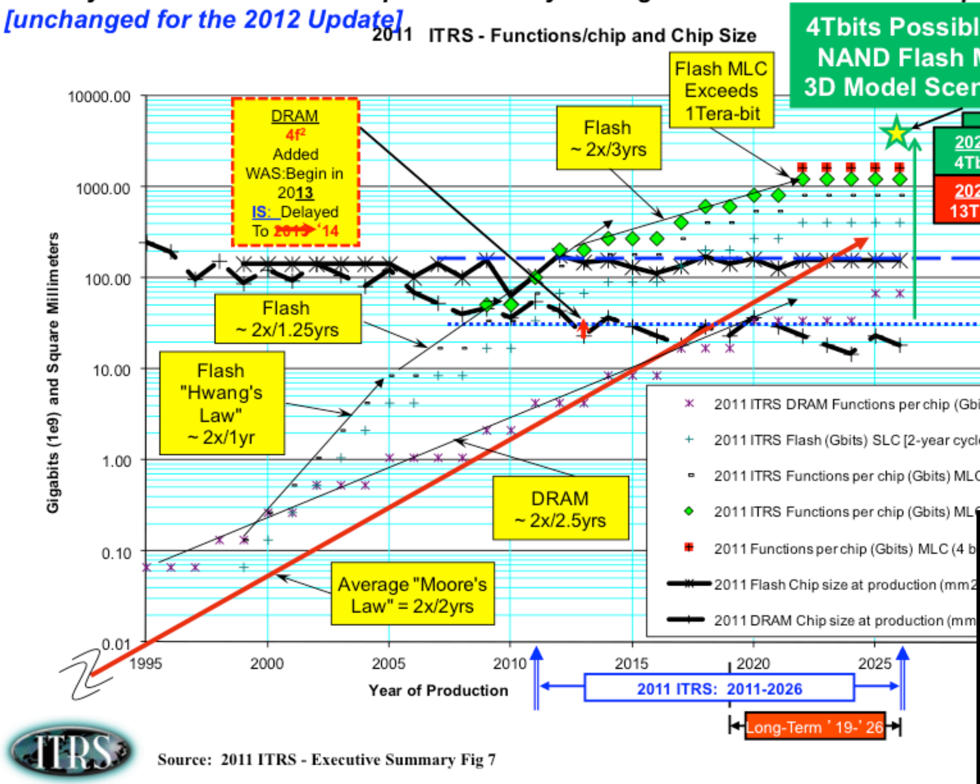


# How has OS design choices been influenced by technological change?

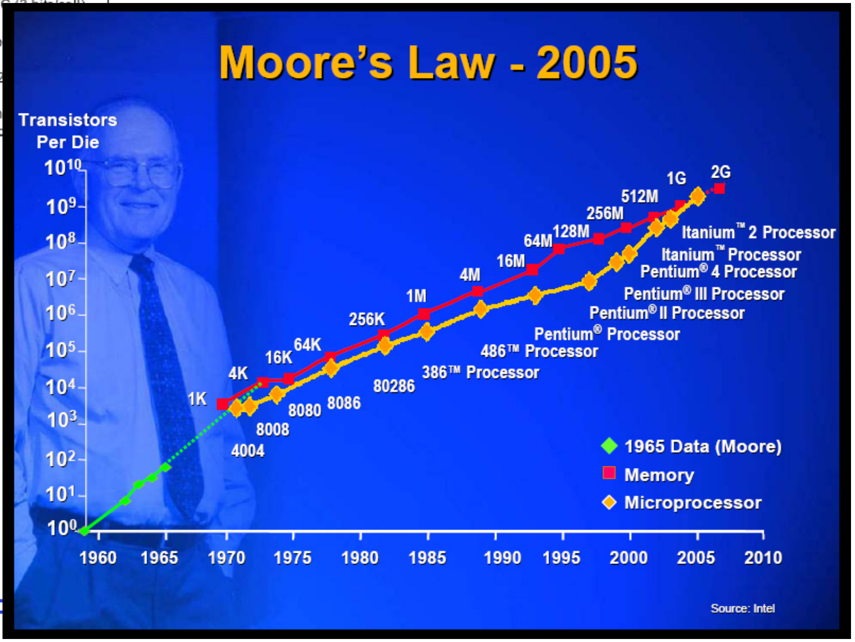


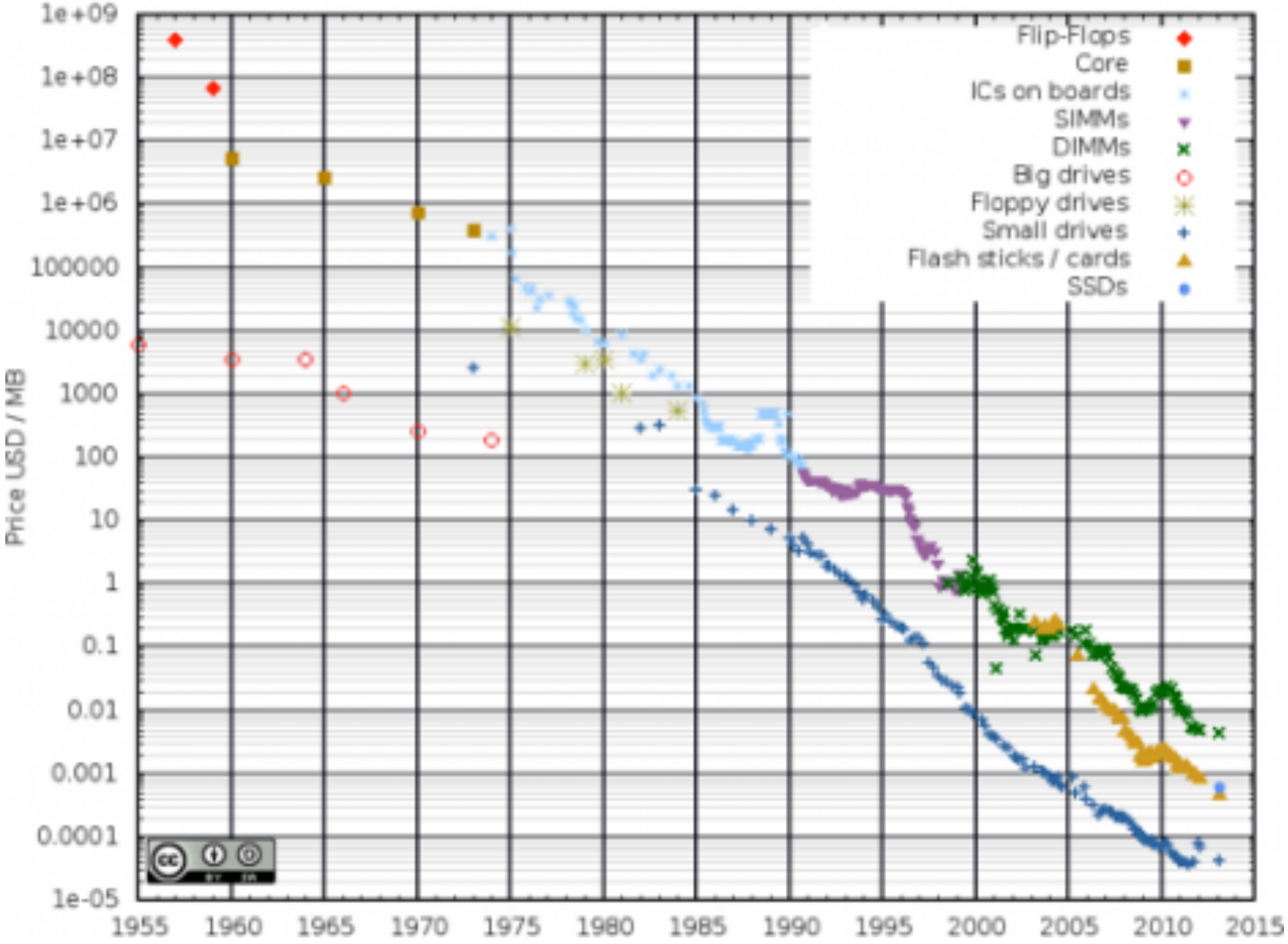
# RAM?

***Figure 7 2011 ITRS Product Technology Trends:***
***Memory Product Functions/Chip and Industry Average "Moore's Law" and Chip Size Trends***
*[unchanged for the 2012 Update]*



Source: 2011 ITRS - Executive Summary Fig 7




Flip-Flops ◆
Core ■
ICs on boards
SIMMs ▼
DIMMs ×
Big drives ○
Floppy drives ✳
Small drives +
Flash sticks / cards ▲
SSDs ●
Price USD / MB
1e+09
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
1e-05
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015

# Summary

- Memory is a resource that must be multiplexed
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources
- Simple Protection through segmentation
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well
- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table: size of page table related to physical memory size