Page Tables, File Systems

23 January 2025
Lecture 12

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

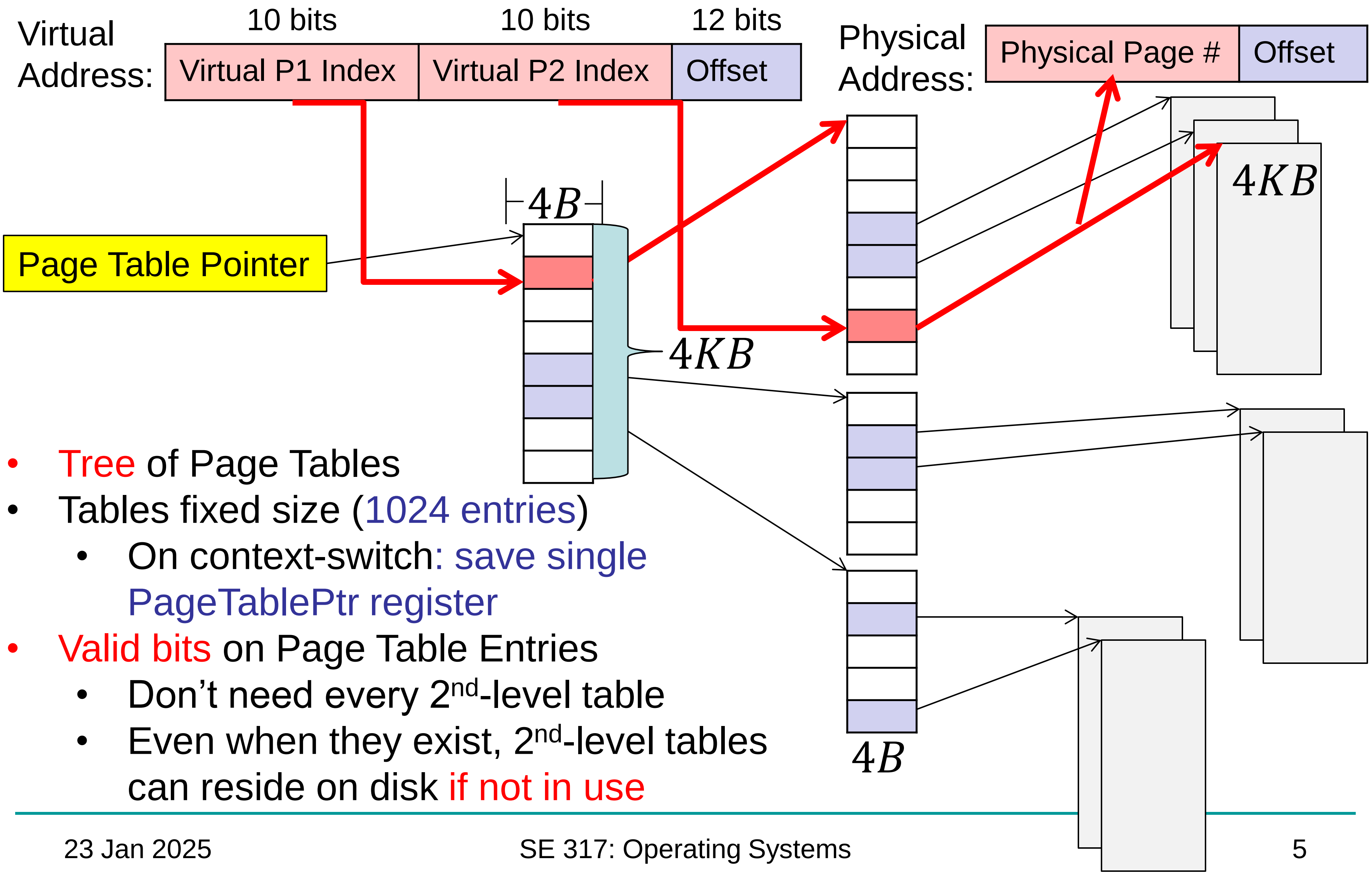
Relocating loader	Segments	Fragmentation • Internal • External	Base and Bound
Offset	Virtual Address	Physical Address	Page Table Entry
Valid/Invalid bit	Page	Page table	Multi-level page table

Topics for Today

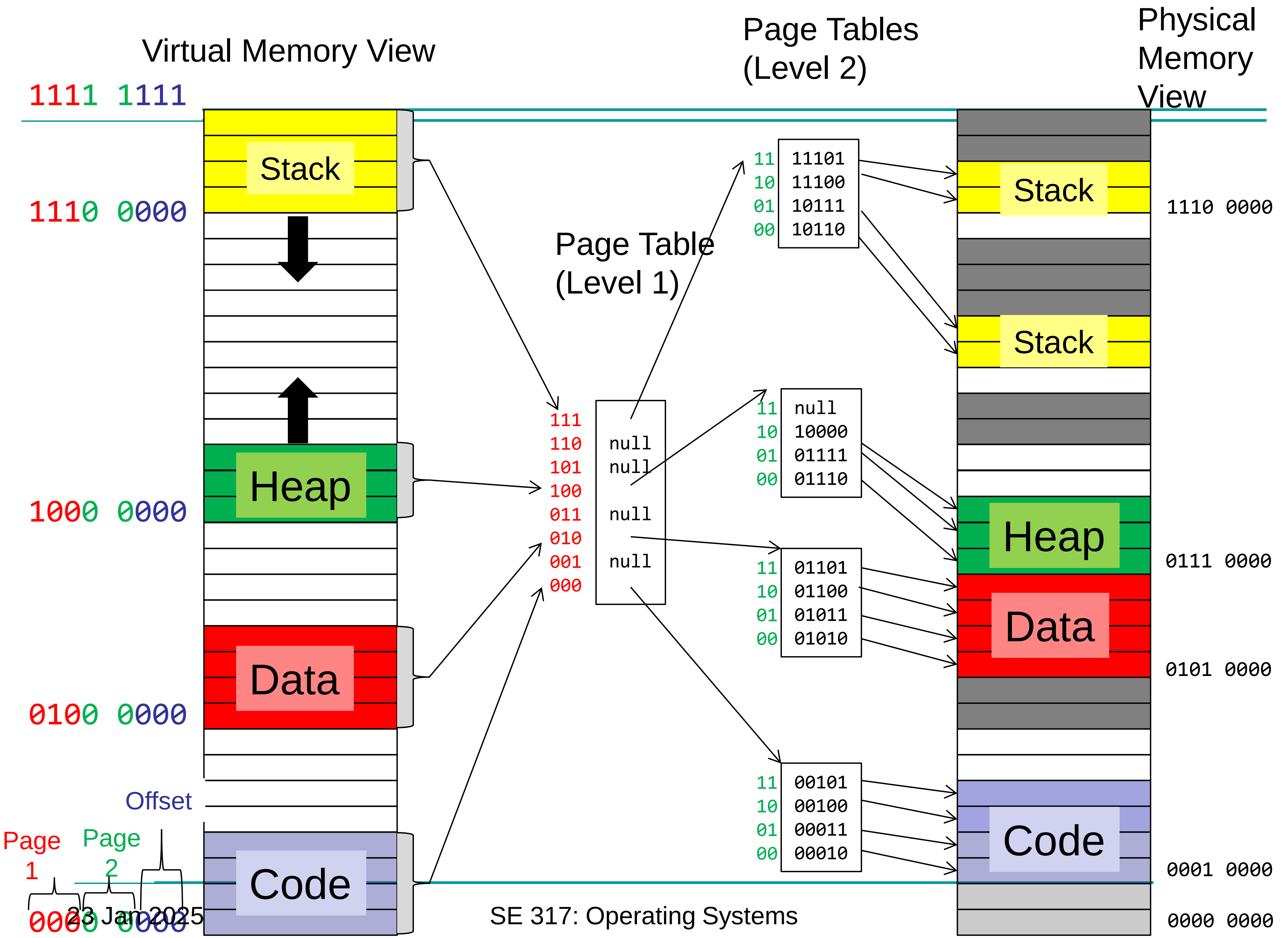
- Page Tables
- File Systems
  - Introduction to File Systems

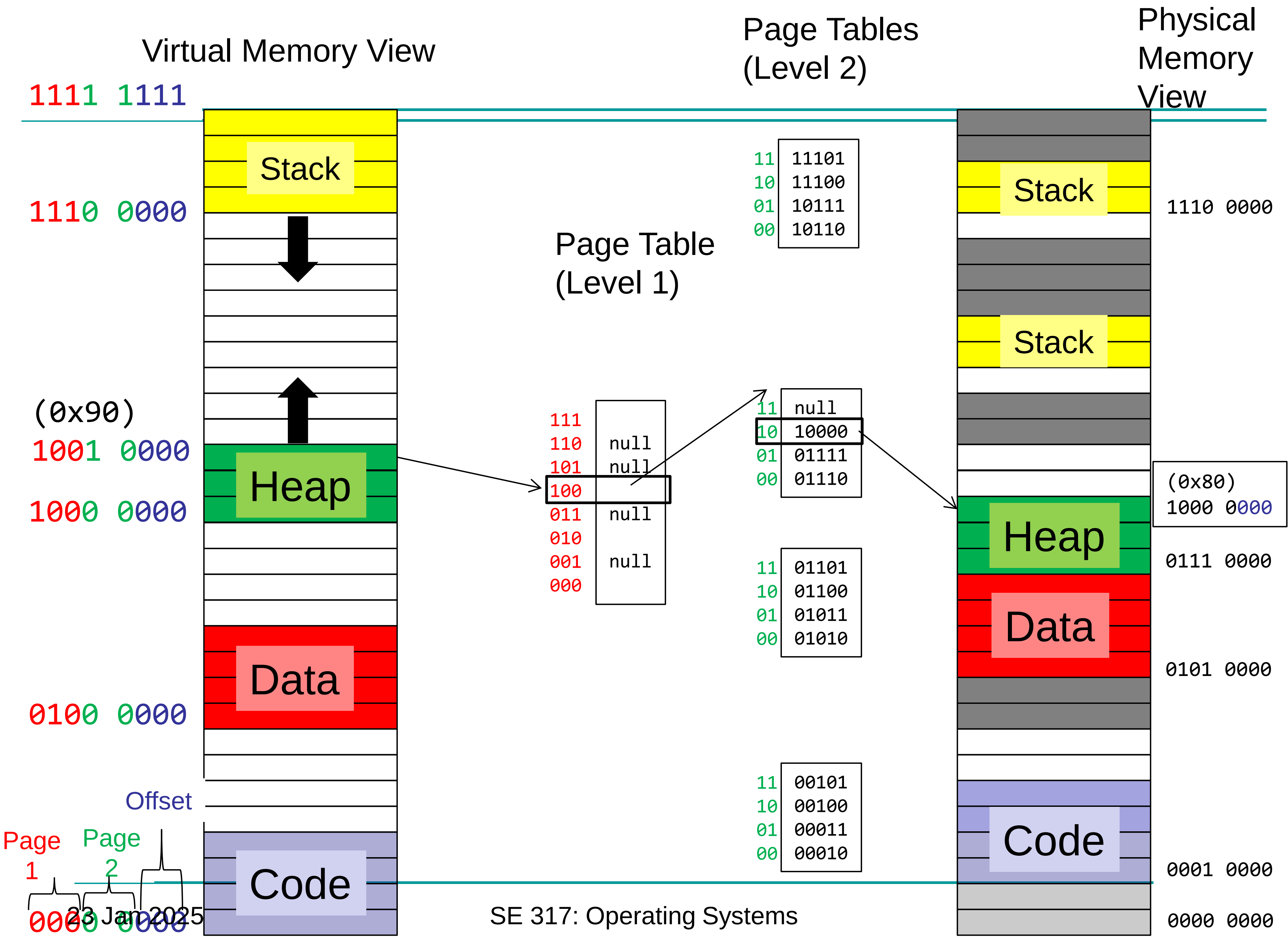
Today's Concepts

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

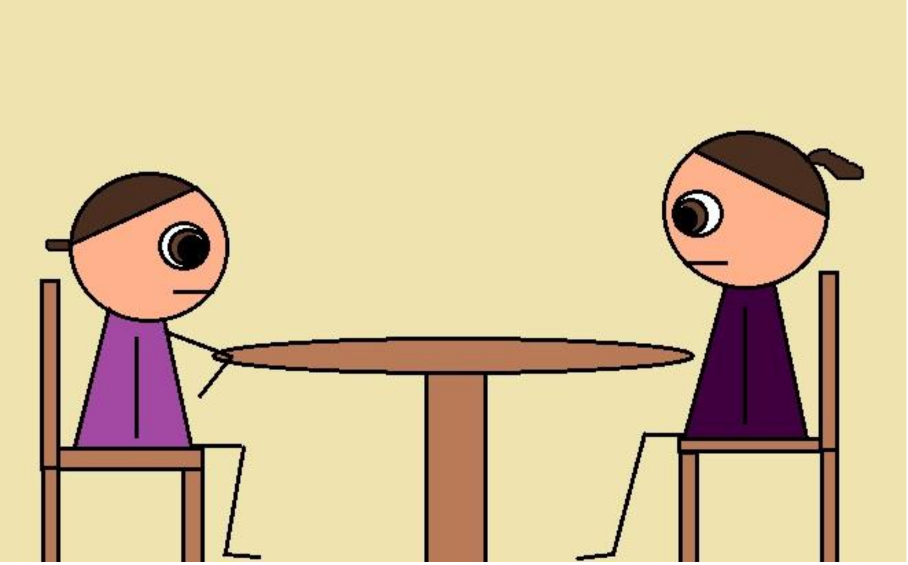

Virtual Memory View
Page Tables (Level 2)
Physical Memory View
Page Table (Level 1)
1111 1111
1110 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
Offset
Page 1
Page 2
111
110
101
100
011
010
001
000
null
null
null
null
11 11101
10 11100
01 10111
00 10110
11 null
10 10000
01 01111
00 01110
11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

Result

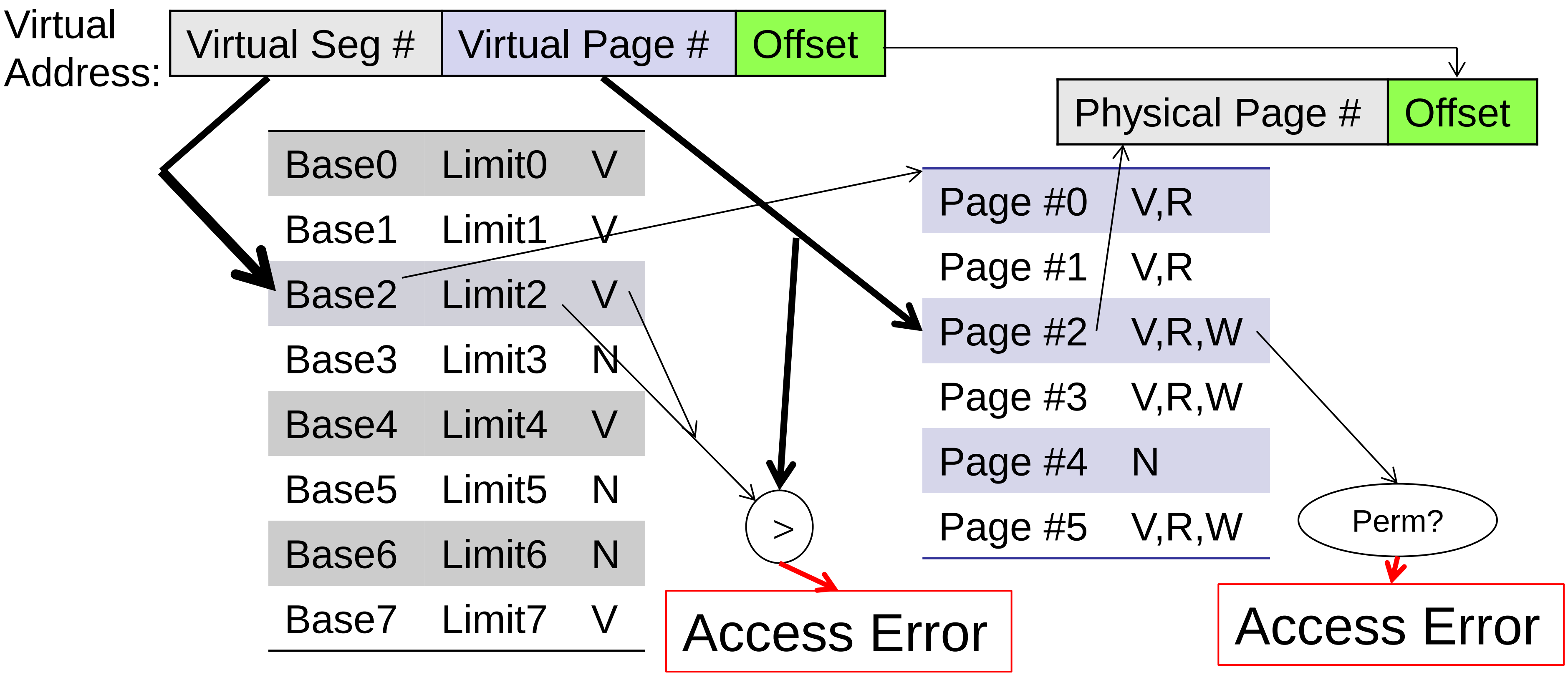
In best case, total size of page tables ≈ number of pages used by program virtual memory.

Requires two additional memory access!



Multi-level Translation: Segments + Pages

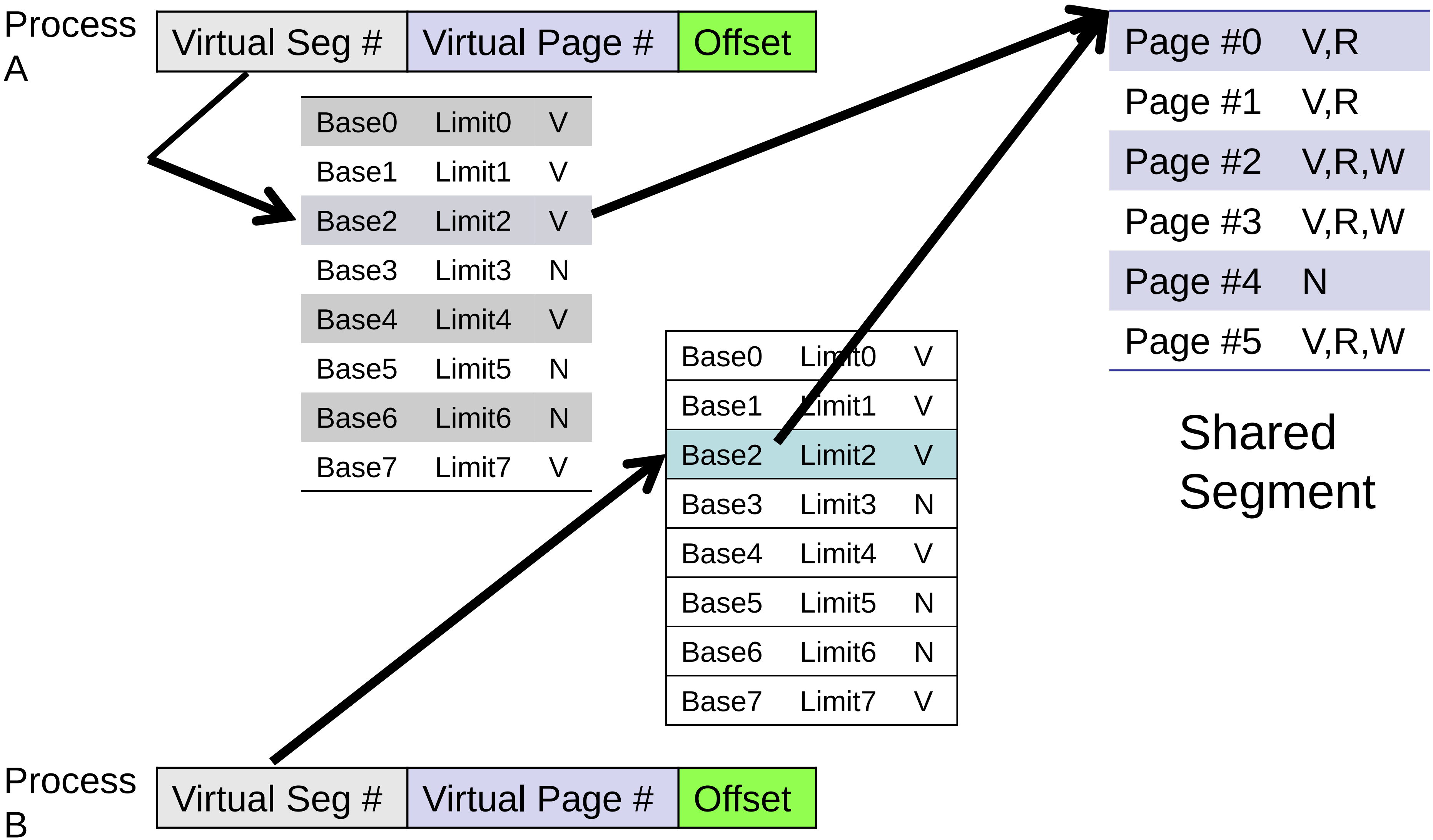
- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



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)?

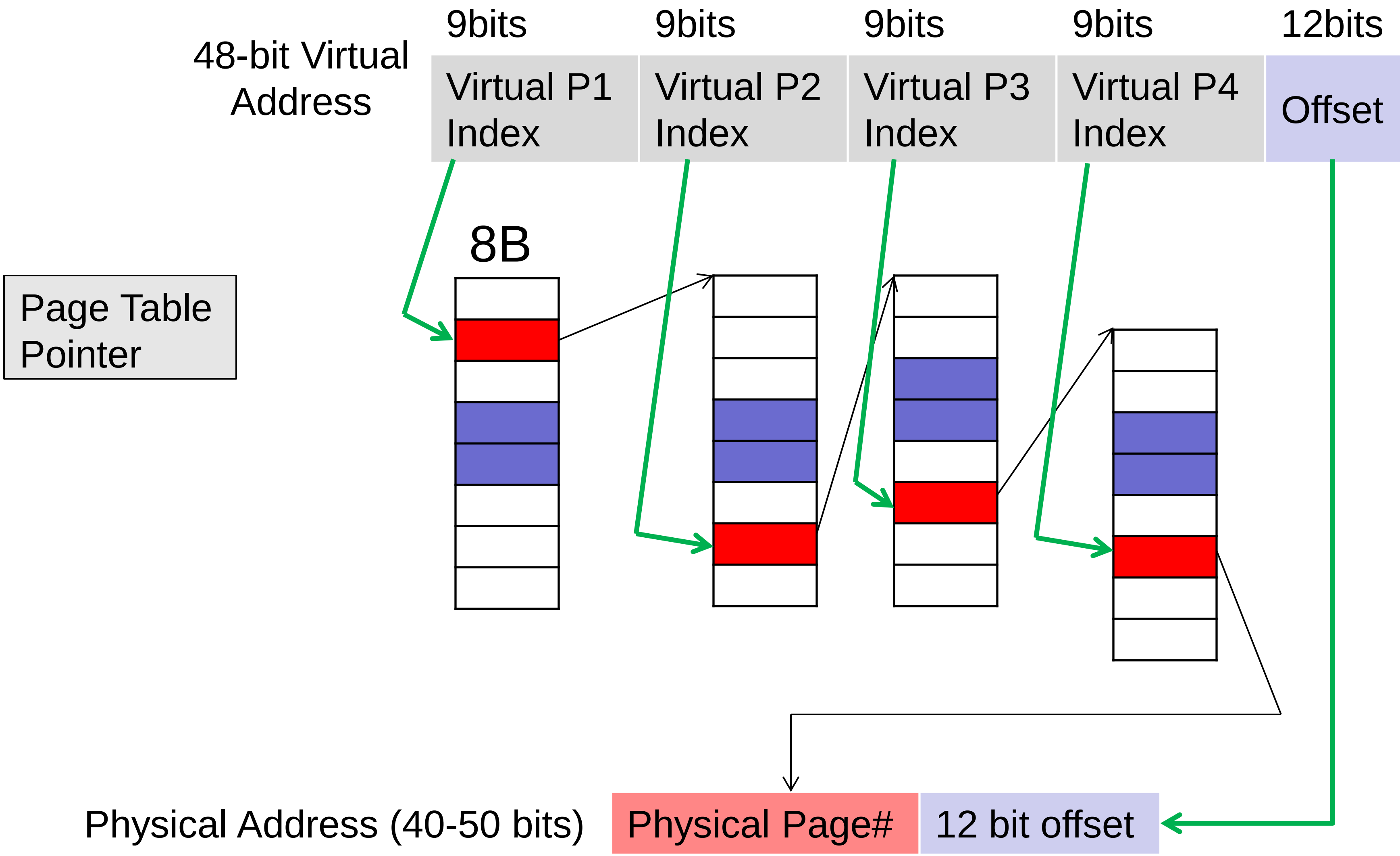


Multi-level Translation Analysis

- 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)

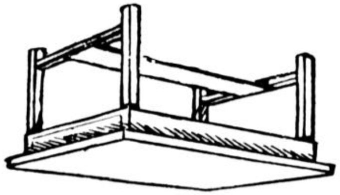
- Cons:
  - One pointer per page (typically $4K - 16K$ pages today)
  - Page tables need to be contiguous
    - However, previous example keeps tables to exactly one page in size
  - Two (or more, if > 2 levels) lookups per reference
    - Seems very expensive!

X86_64: Four-level page table!



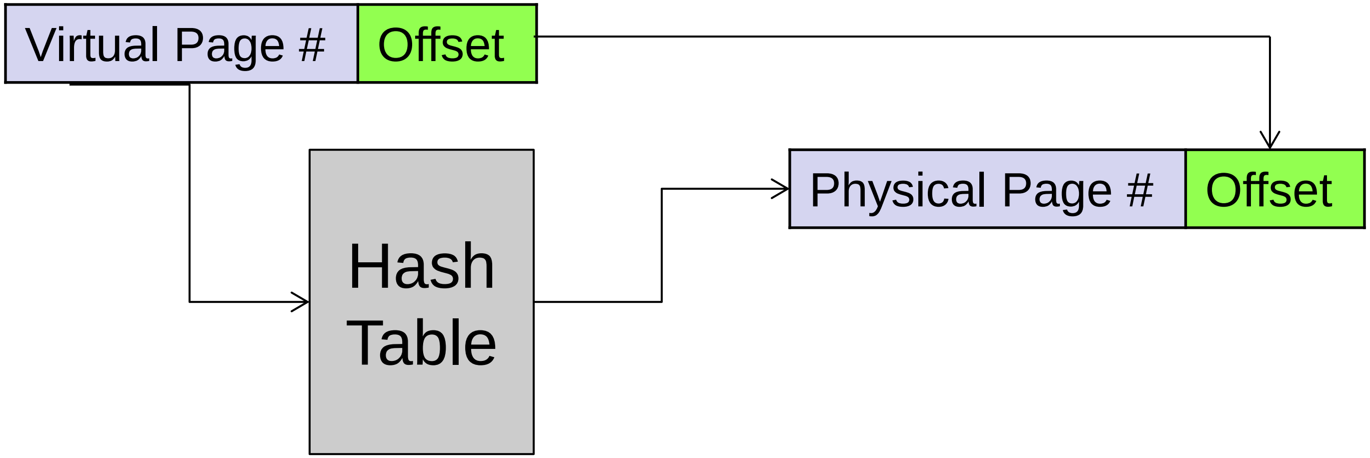
Any other ideas to make it smaller?

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

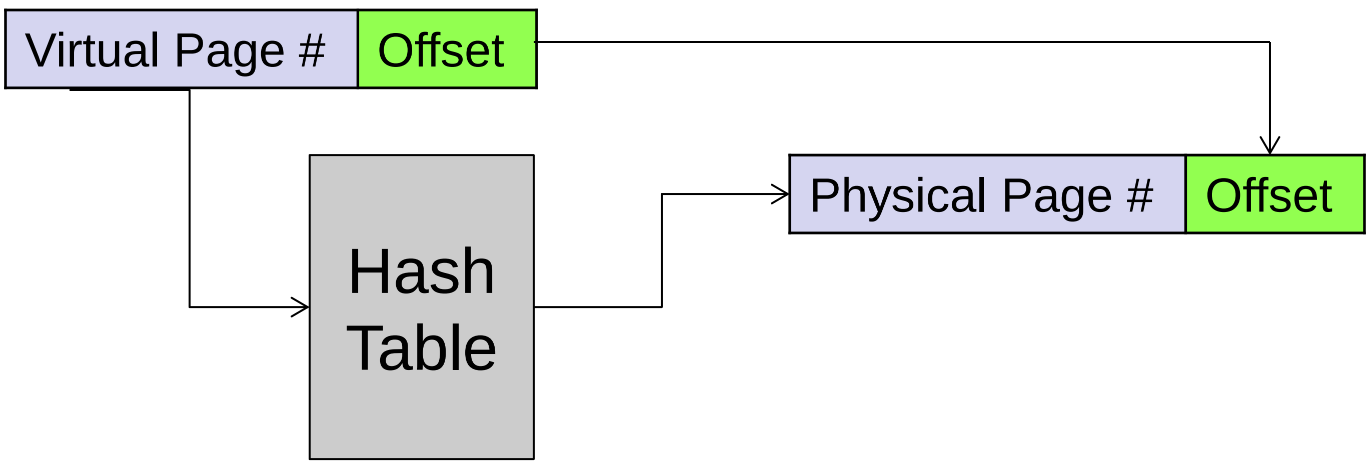
- Answer: use a hash table
  - Called an "Inverted Page Table"



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

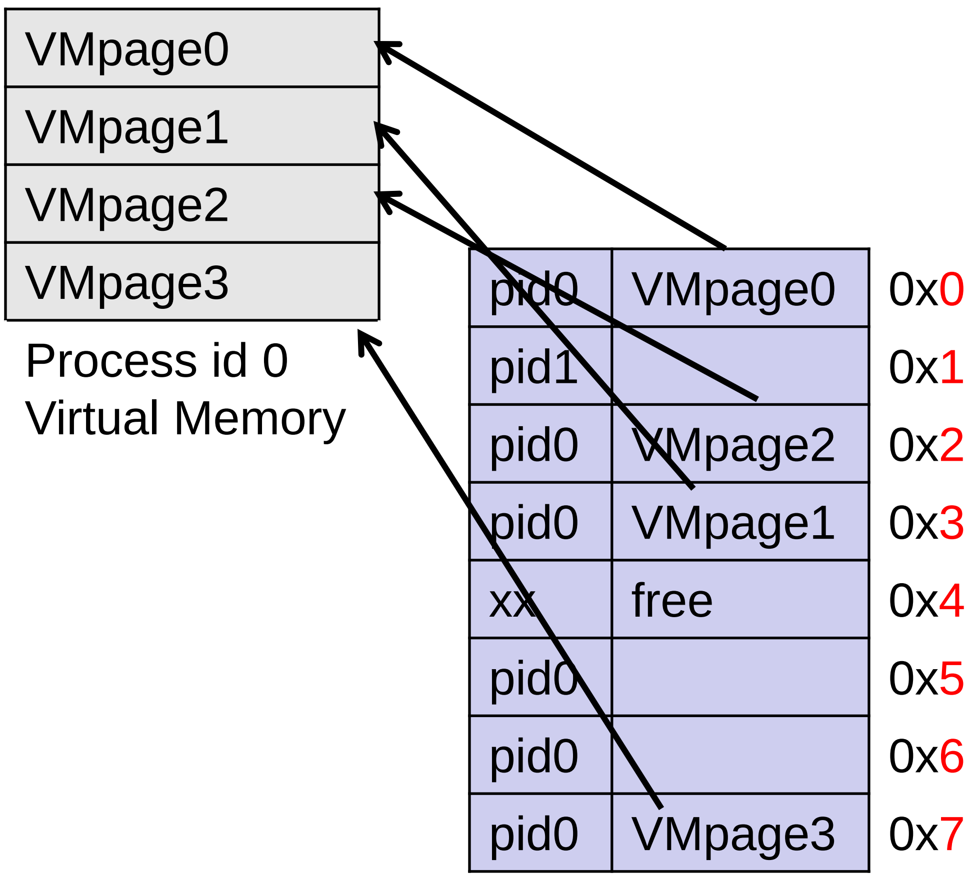
No!

Too Slow!

Too many almost empty tables

IA64: Inverse Page Table (IPT)

Idea: Index page table by physical pages instead of VM



Inverse Page Table

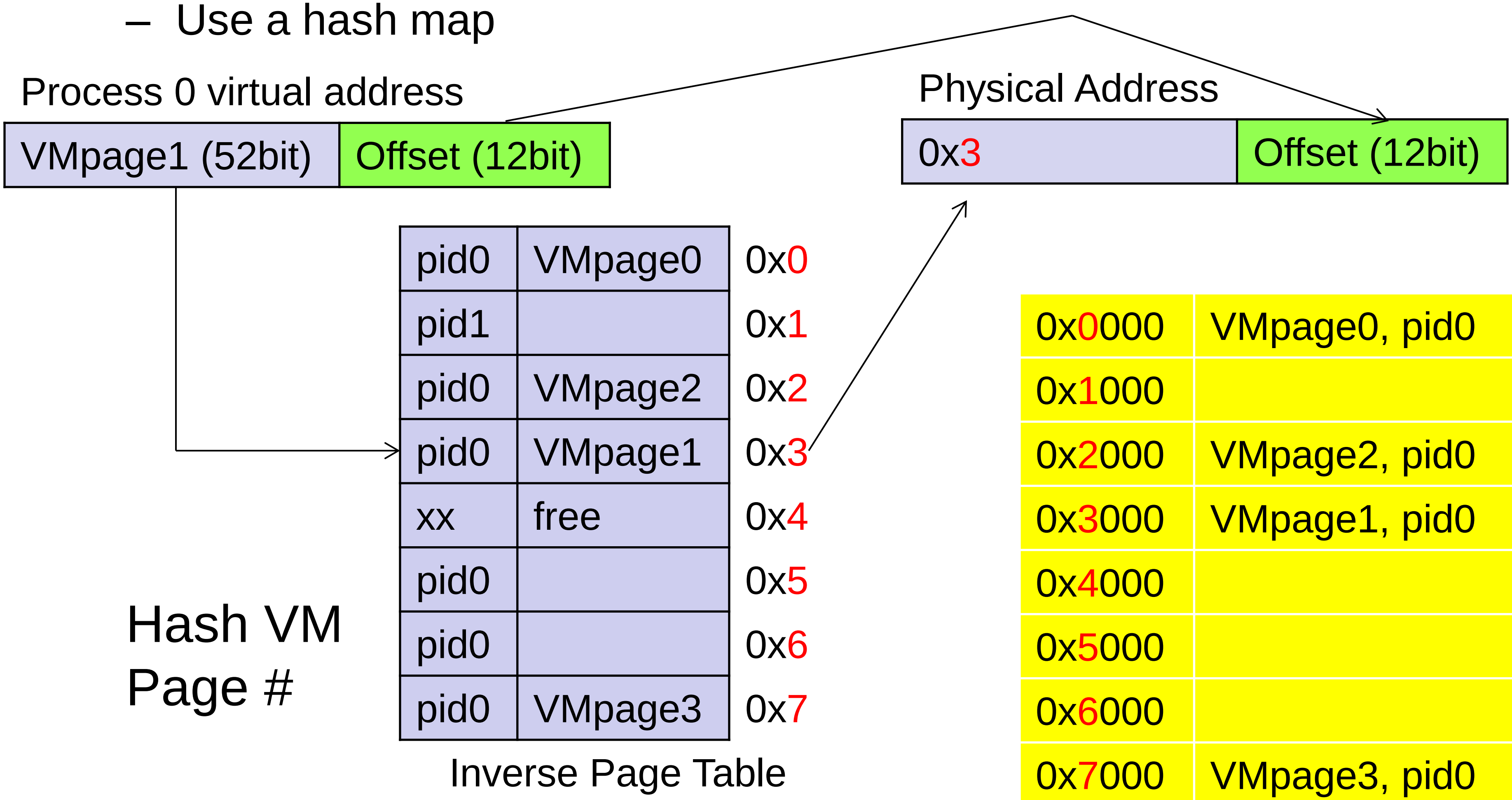
0x0000	VMpage0, pid0
0x1000	
0x2000	VMpage2, pid0
0x3000	VMpage1, pid0
0x4000	
0x5000	
0x6000	
0x7000	VMpage3, pid0

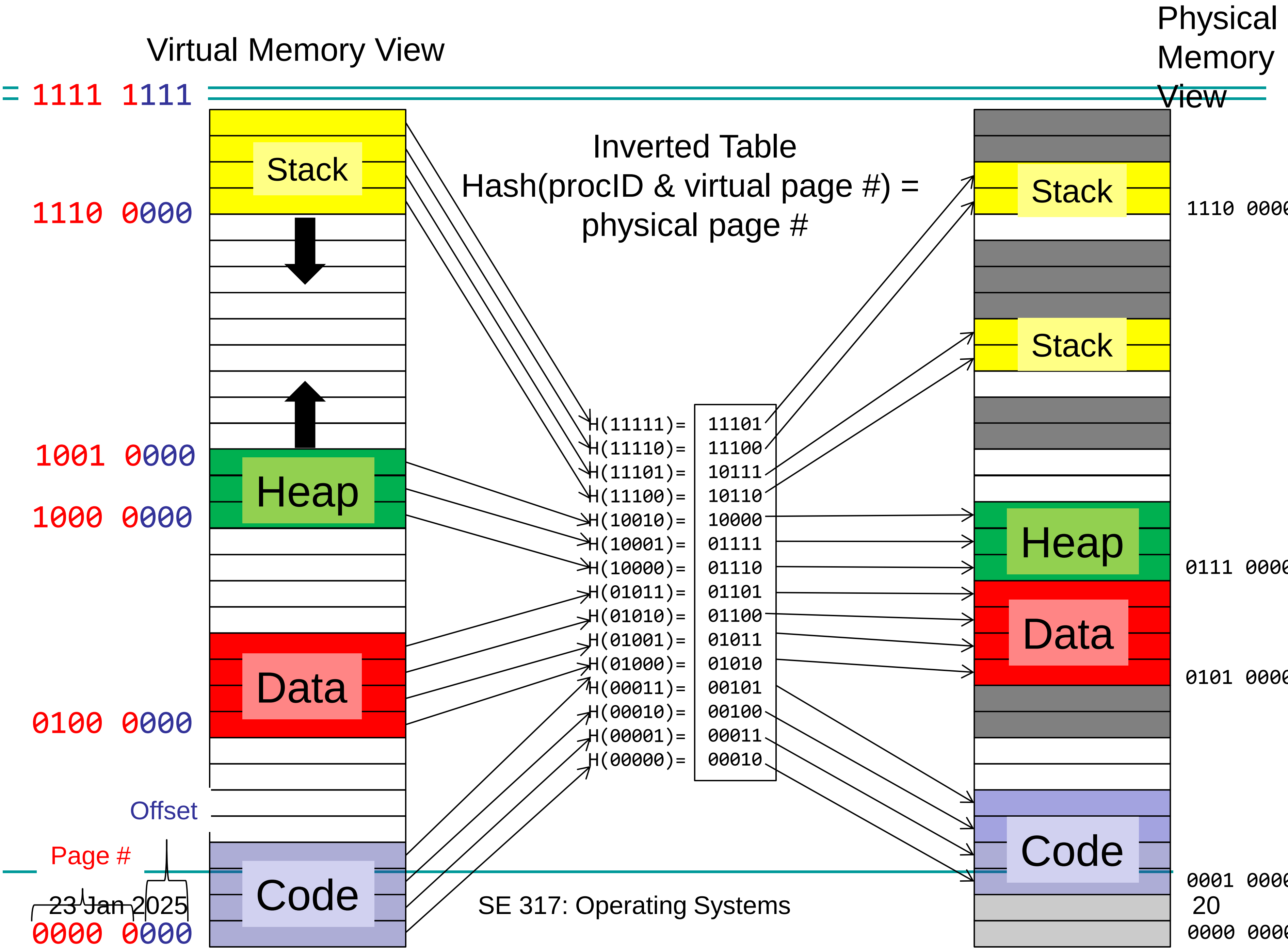
Physical Memory in 4KB pages

Page numbers in red

IPT address translation

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



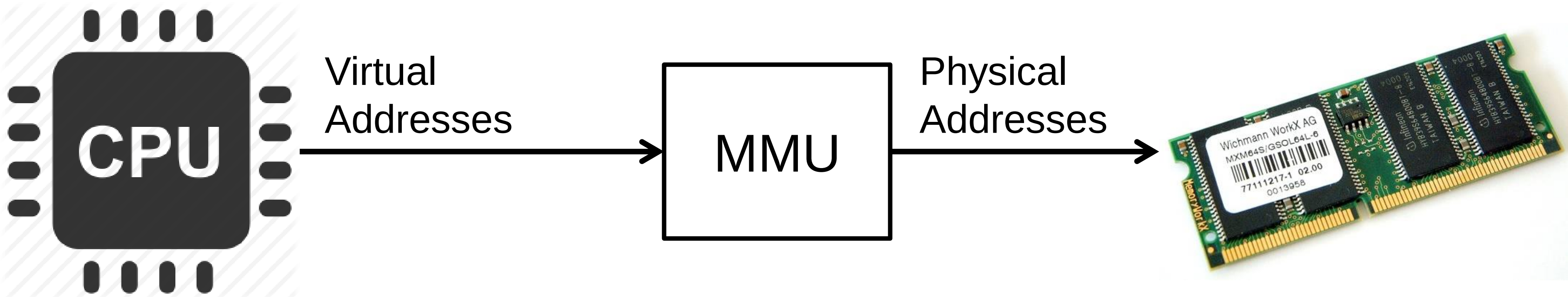


Virtual Memory View
Physical Memory View
1111 1111
1110 0000
1001 0000
1000 0000
0100 0000
0000 0000
Stack
Heap
Data
Code
Offset
Page #
Inverted Table
Hash(procID & virtual page #) = physical page #
H(11111)= 11101
H(11110)= 11100
H(11101)= 10111
H(11100)= 10110
H(10010)= 10000
H(10001)= 01111
H(10000)= 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
1110 0000
0111 0000
0101 0000
0001 0000
0000 0000

Address Translation Comparison

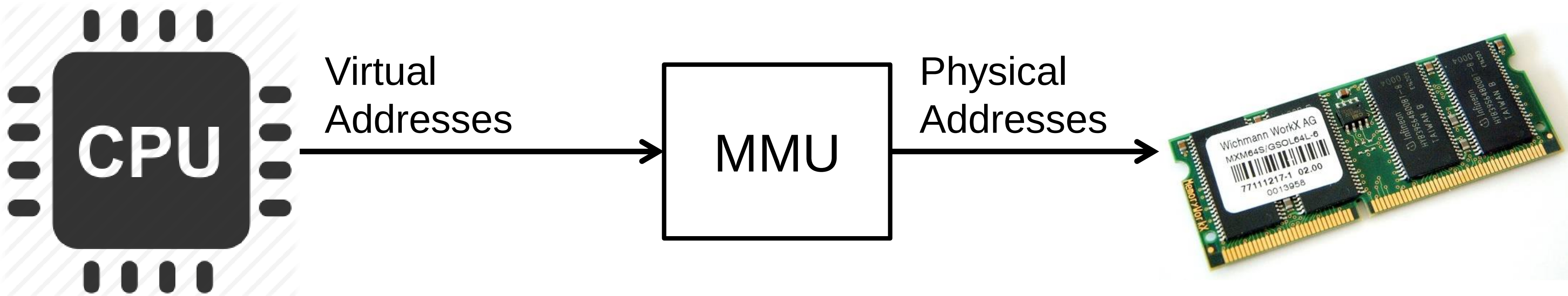
	Advantages	Disadvantages
Simple Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation Two-level pages	Table size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

So Far

- Page Tables
- File Systems
  - Introduction to File Systems

Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

  – File System Components

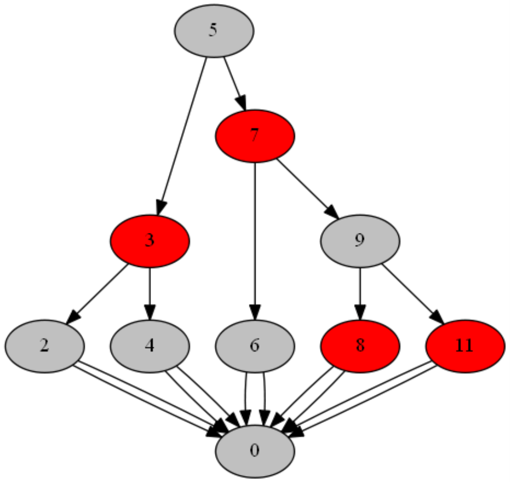
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks

  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc

User vs. System View of a File

User's view:

- Durable Data Structures

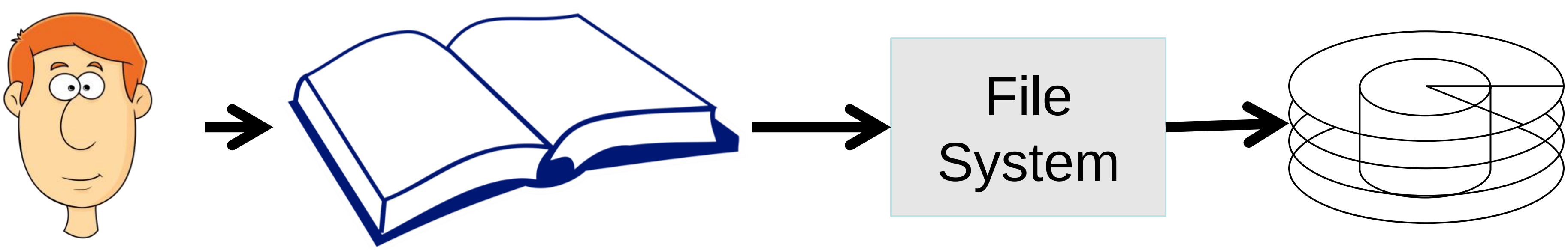


System's View:

- System call interface:
  - Collection of Bytes (UNIX)
  - Doesn't matter to system what kind of data structures you want to store on disk!
- Inside OS:
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size $\geq$ sector size; in UNIX, block size is 4KB

Translating from User to System View

- What happens if user says: *Give me bytes 2-12?*
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: *Write bytes 2-12?*
  - Fetch block, Modify portion, Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks



So you are going to design a file system …

What factors are critical to the design choices?

Durable data store → It's all on disk

↓

Disk Performance!

Maximize sequential access, minimize seeks

↓

Open before Read/Write

Can perform protection checks and look up where the actual file resource are, in advance

So you are going to design a file system …

Size is determined as files are used

- Can write (or read zeros) to expand the file
- Start small and grow, need to make room

↓

Organize into directories

What data structure (on disk) do we use for that?

↓

Need to allocate and free blocks

Keep access efficient

Defragmenting

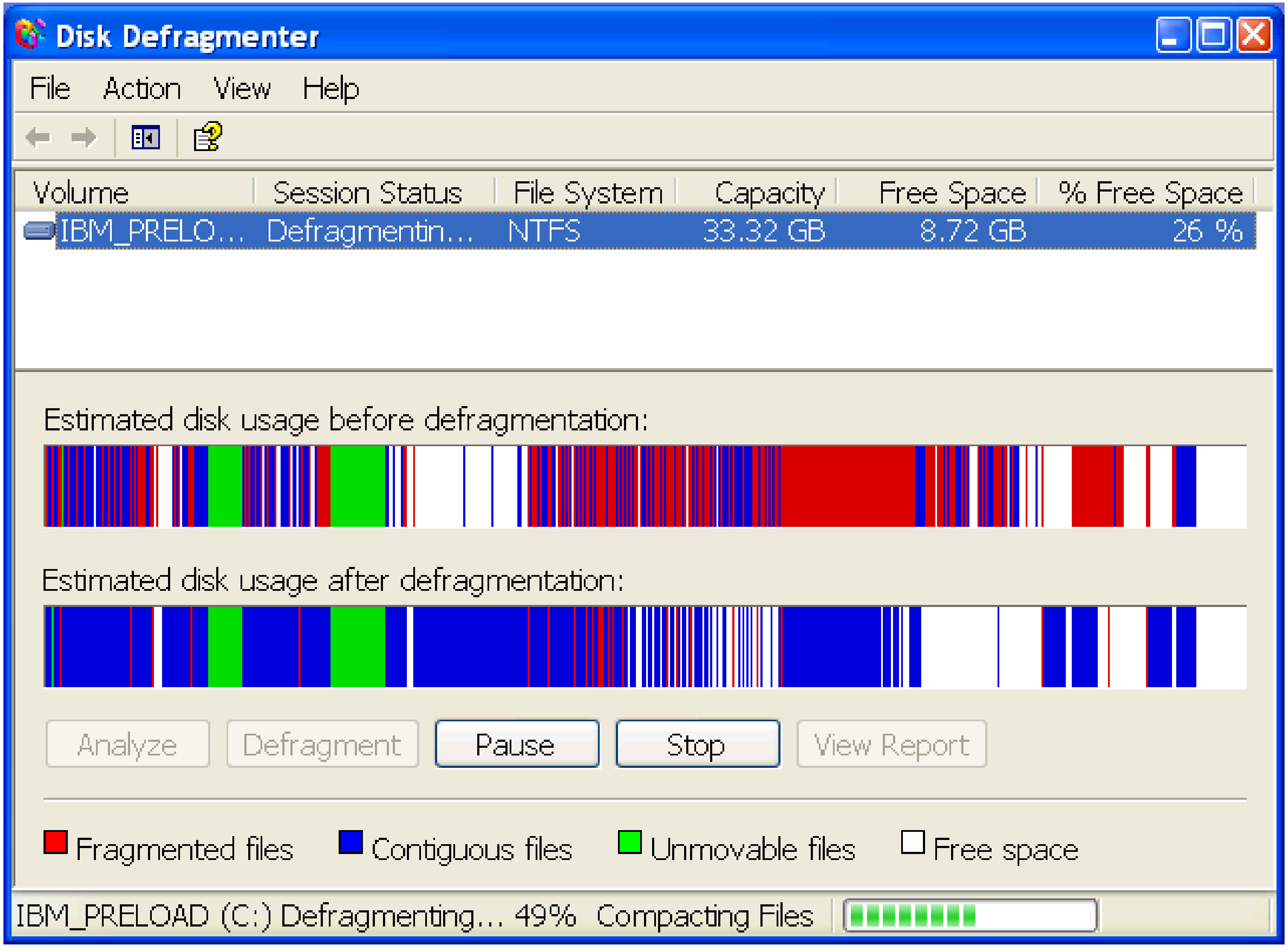


Image source: http://blog.shane-smith.com/blog:116

Disk Management Policies

- Basic entities on a disk:
  - File: User-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files

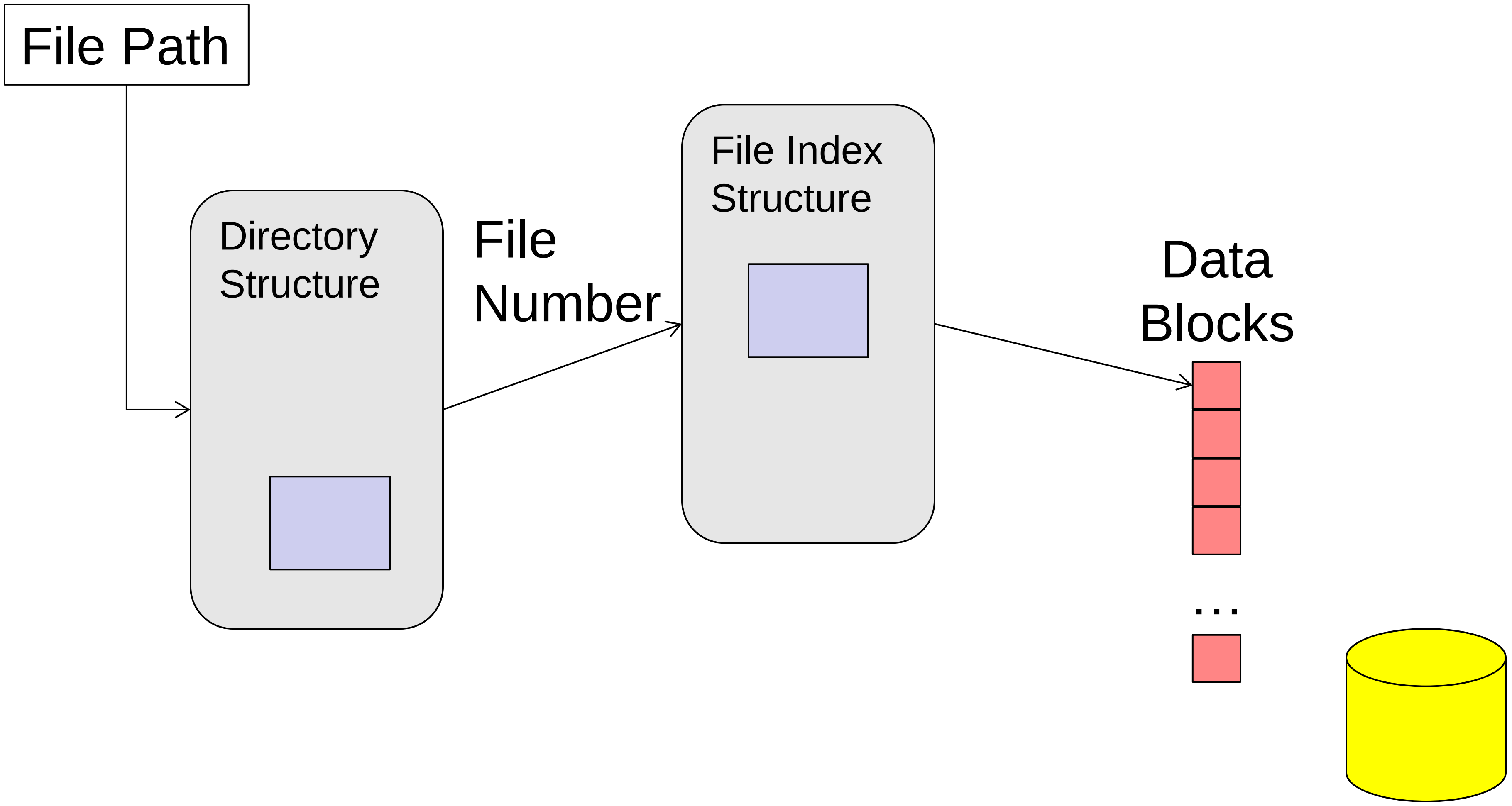
- Access disk as linear array of sectors.
  Two Options:
  1. Identify sectors as vectors `[cylinder, surface, sector]`. Sort in cylinder-major order. Not used much anymore.
  2. Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.

- Controller translates from address⇒physical position
  - First case: OS/BIOS must deal with bad sectors
  - Second case: Hardware shields OS from structure of disk

Disk Management Policies

- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- Need a way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

Components of a File System

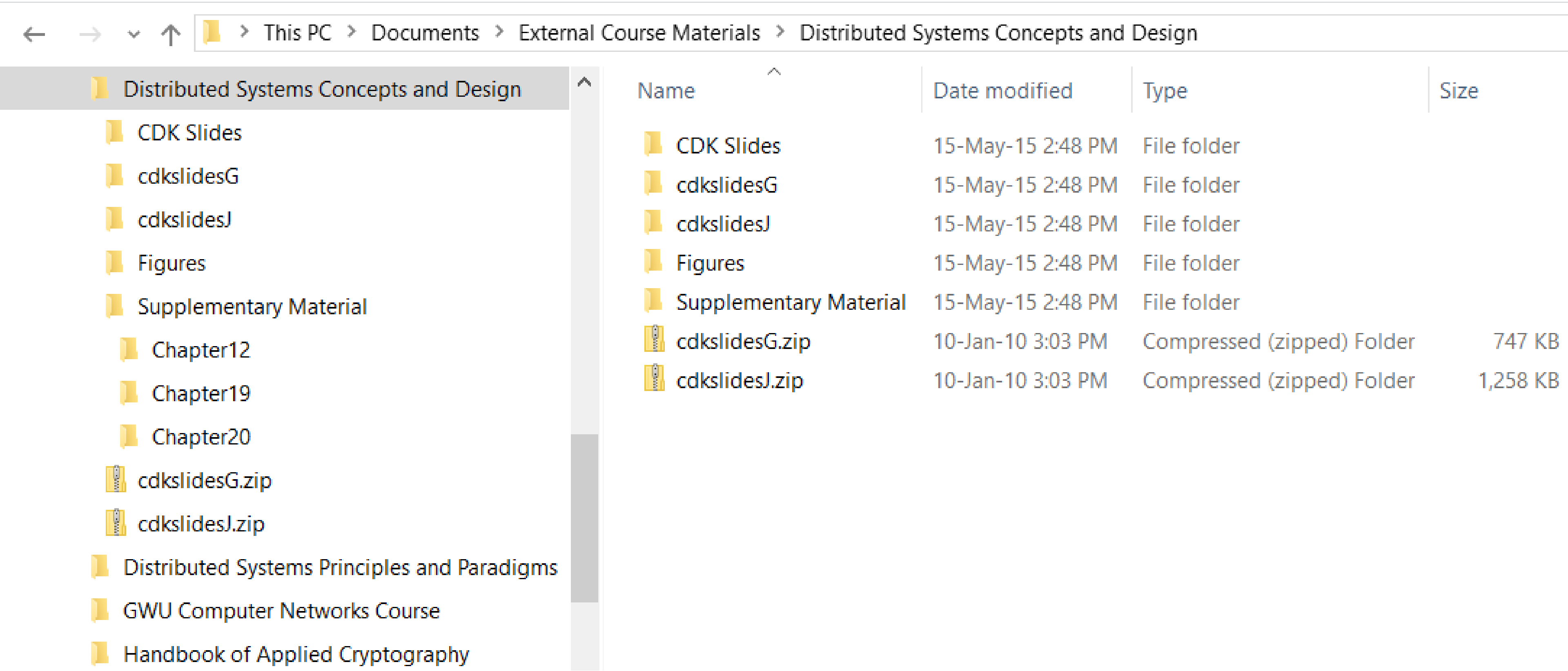


Components of a file system

- Open performs *name resolution*
  - Translates pathname into a “file number”
    - Used as an “index” to locate the blocks
  - Creates a file descriptor in PCB within kernel
  - Returns a “handle” (another `int`) to user process
- Read, Write, Seek, and Sync operate on handle
  - Mapped to **descriptor** and to **blocks**

$$\begin{array}{c}\text{File Name}\\ \text{Offset}\end{array} \xrightarrow{\text{Directory}} \begin{array}{c}\text{File Number}\\ \text{Offset}\end{array} \xrightarrow[\text{Structure}]{\text{Index}} \begin{array}{c}\text{Storage}\\ \text{Block}\end{array}$$

Directories

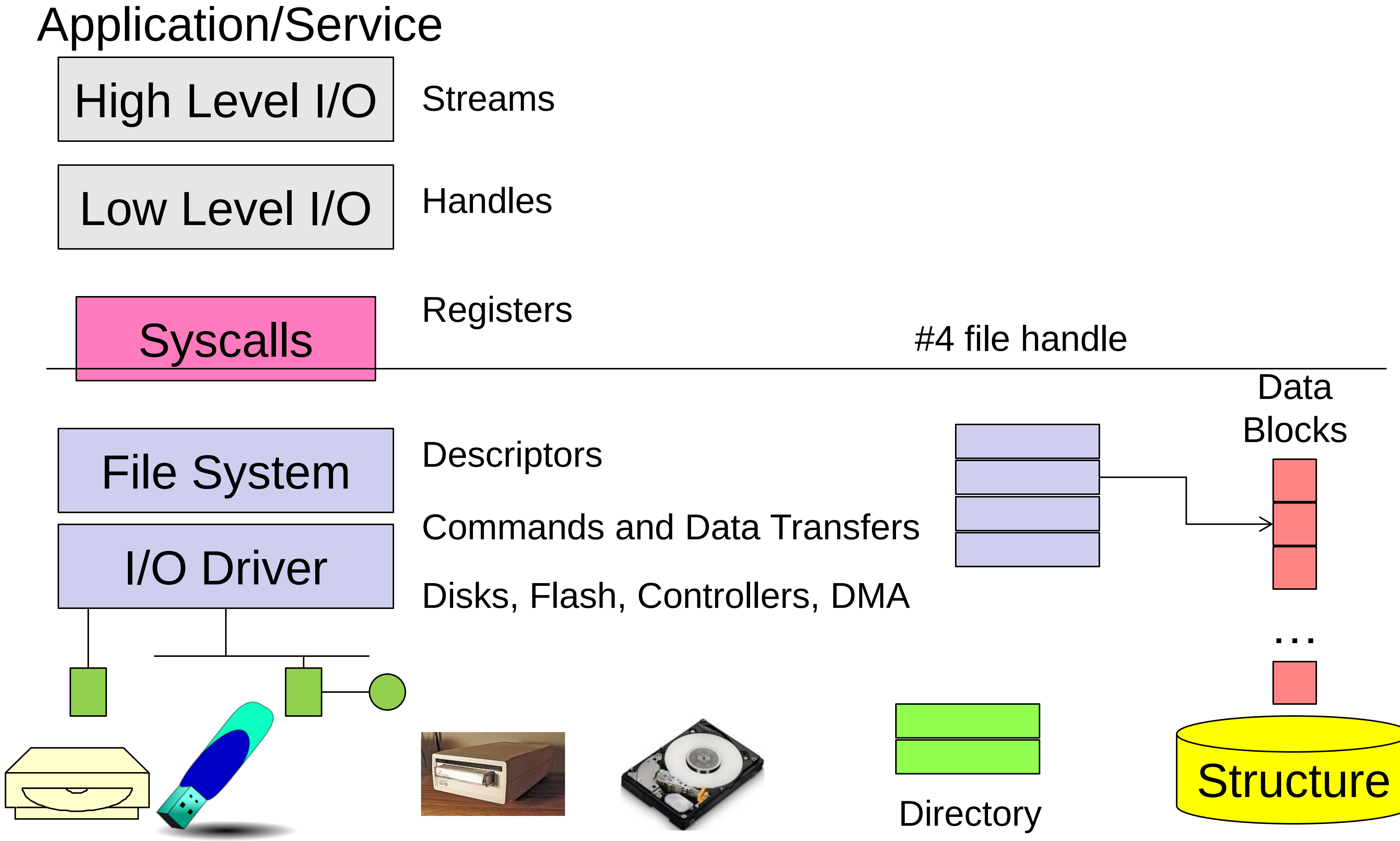


Directory

- Basically a hierarchical structure
- Each directory entry is a collection of
  - Files
  - Directories
    - A link to another entries
- Each has a name and attributes
  - Files have data
- Links (hard links) make it a DAG, not just a tree
  - Soft links (aliases) are another name for an entry

sdcard 2013-12-10 23:51 lrwxrwxrwx -> /mnt/sdcard

I/O & Storage Levels



File

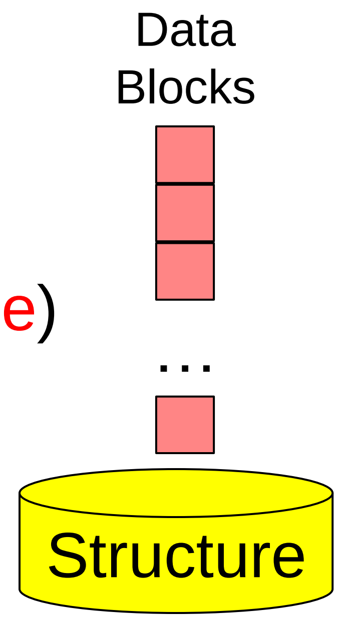
- Named permanent storage

Contains

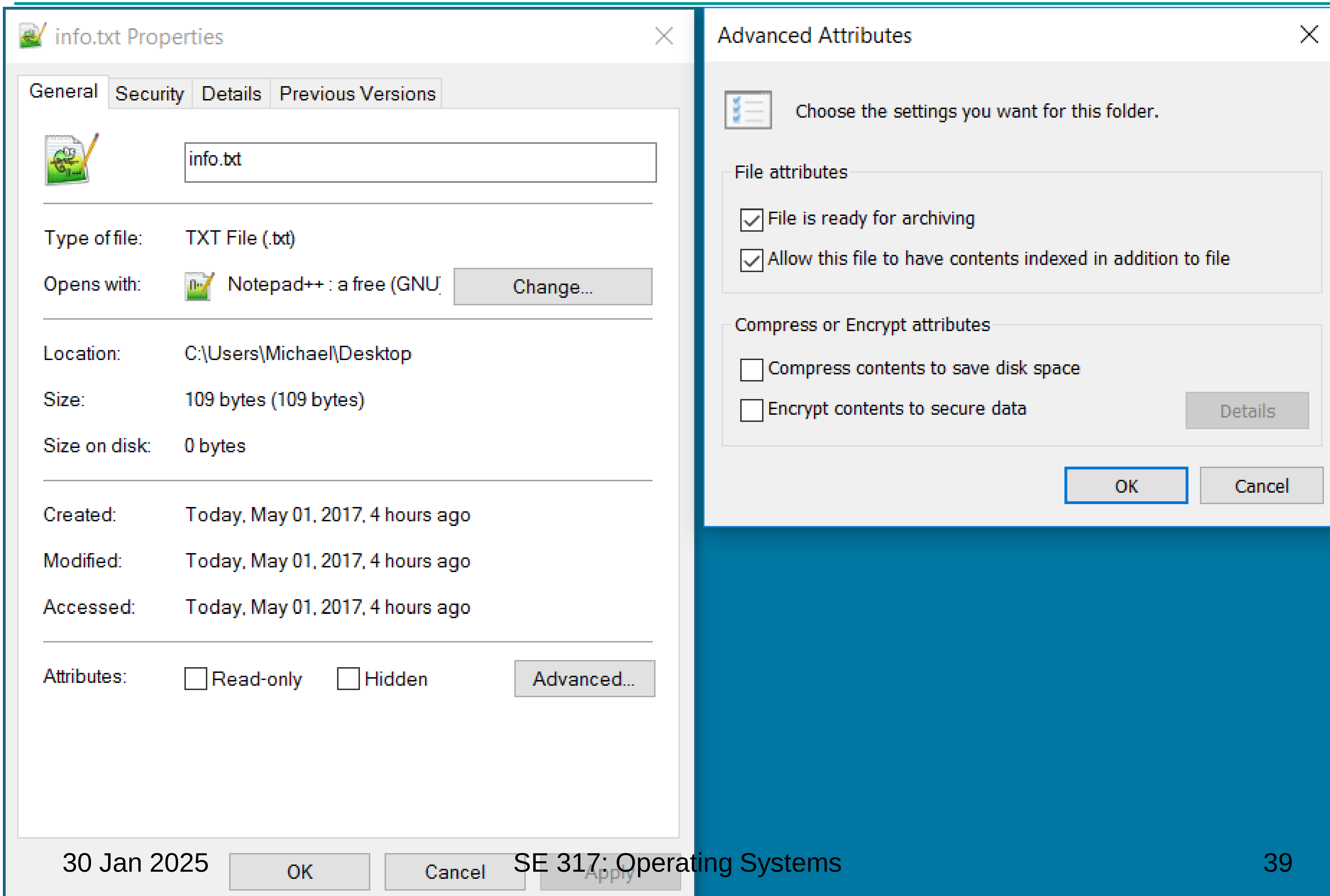
- Data
  - Blocks on disk somewhere
- Metadata (Attributes)
- Owner, size, last opened, …
- Access rights
  - R, W, X
  - Owner, Group, Other (in Unix systems)
  - Access control list in Windows system

File Handle

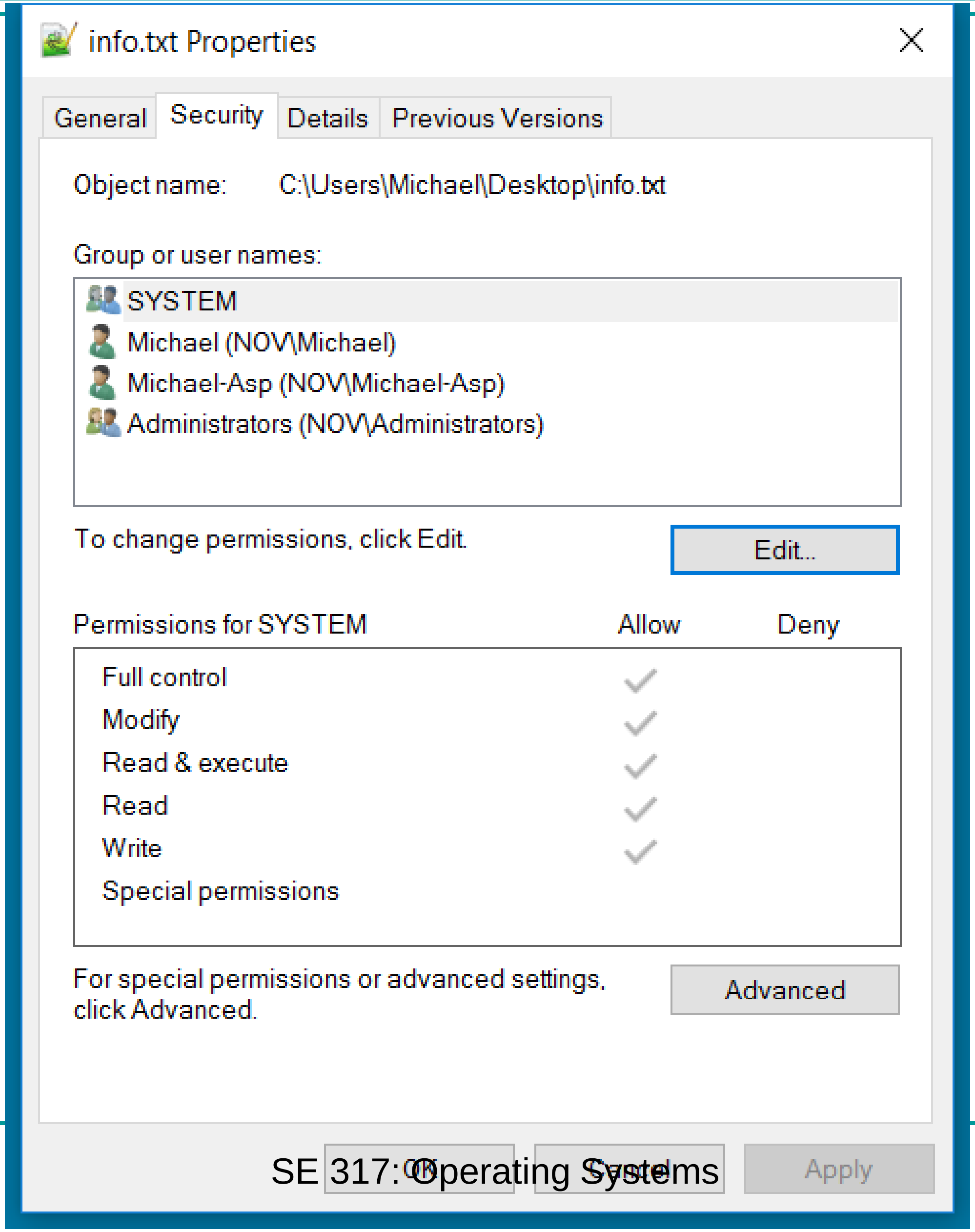
File Descriptor
FileObject (inode)
Position



File Attributes



File Attributes



File Metadata

alice_promo.mp4 Properties

General | Security | Details | Previous Versions

Property	Value
Description	
Title	
Subtitle	
Rating	☆ ☆ ☆ ☆ ☆
Tags	
Comments	
Video	
Length	00:03:59
Frame width	352
Frame height	288
Data rate	245kbps
Total bitrate	364kbps
Frame rate	29 frames/second
Audio	
Bit rate	118kbps
Channels	2 (stereo)
Audio sample rate	44 kHz
Media	

Remove Properties and Personal Information

OK | Cancel | Apply

alice_promo.mp4 Properties

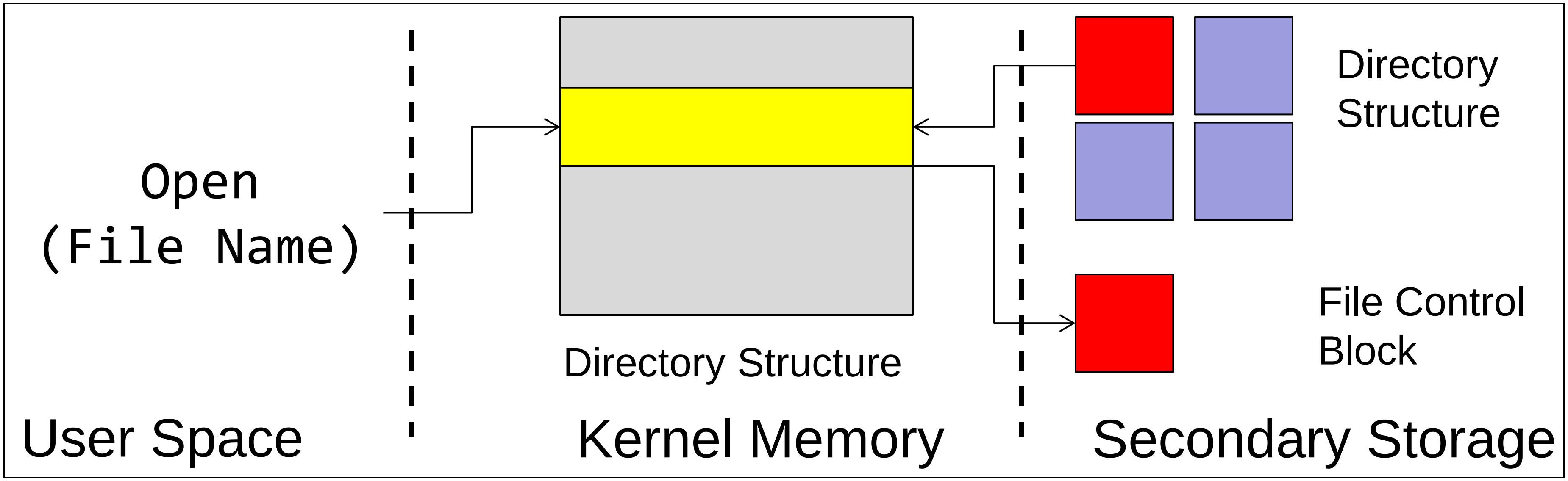
General | Security | Details | Previous Versions

Property	Value
Mood	
Part of set	
Initial key	
Beats-per-minute	
Protected	No
File	
Name	alice_promo.mp4
Item type	MP4 File
Folder path	C:\Users\Michael\Documents\Extern...
Size	10.5 MB
Date created	03-Jun-13 3:48 PM
Date modified	03-Jun-13 3:56 PM
Attributes	A
Availability	Available offline
Offline status	
Shared with	Michael-Asp
Owner	NOV\Michael
Computer	NOV (this PC)

Remove Properties and Personal Information

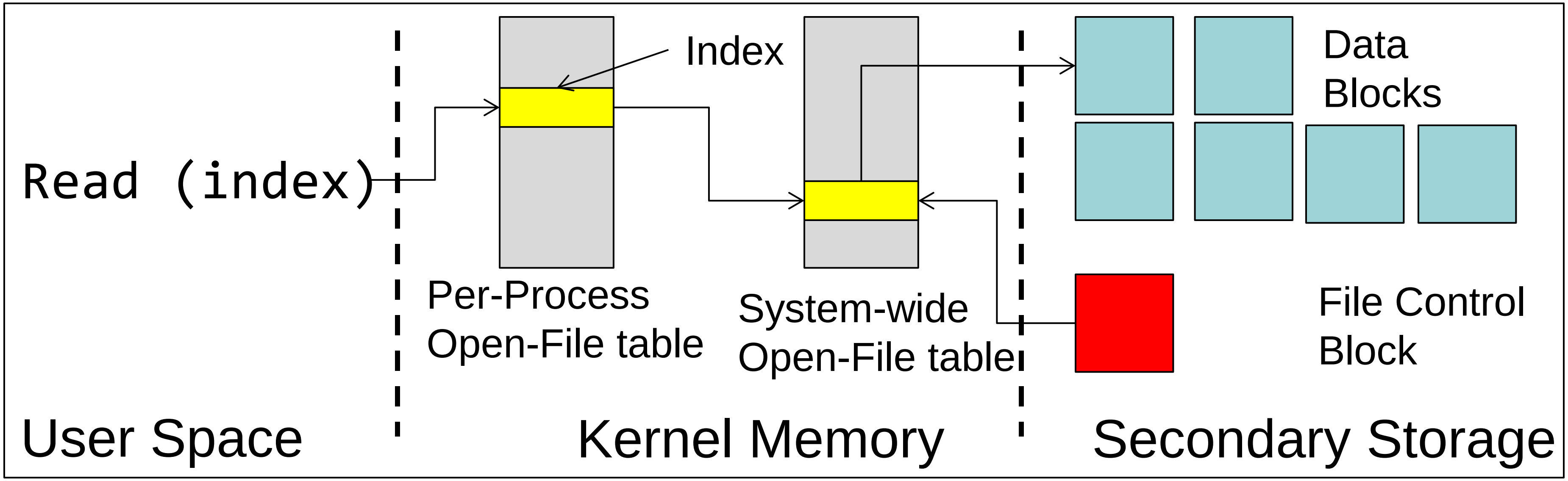
OK | Cancel | Apply

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called "file handle") in open-file table

In-Memory File System Structures



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

Conclusion

- Page Tables
- File Systems
  - Introduction to File Systems