# CS162
## Operating Systems and Systems Programming
## Lecture 20

## Filesystems 2: Filesystem Case Studies (Con't), Buffering
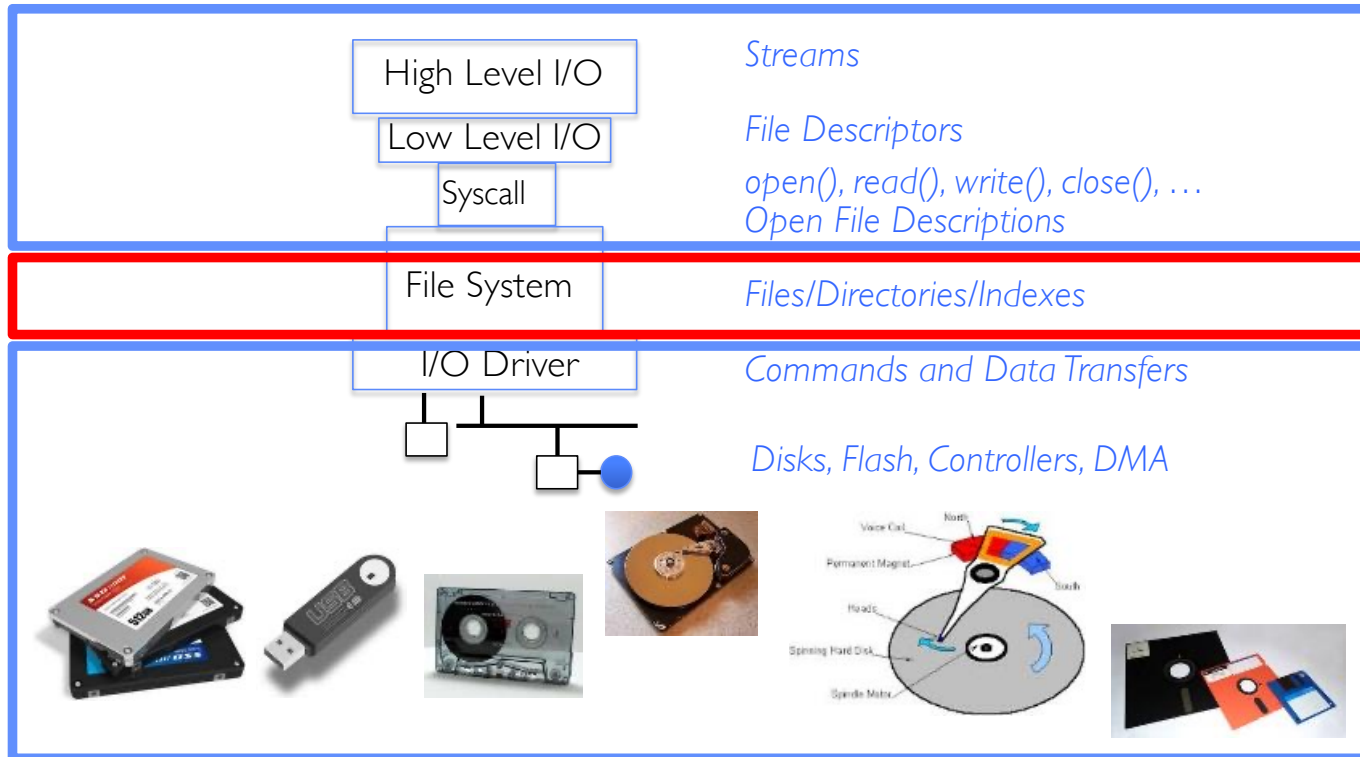
April 8th, 2021

Profs. Natacha Crooks and Anthony D. Joseph

http://cs162.eecs.Berkeley.edu

# Recall: I/O and Storage Layers

Application / Service

| | | |
|---|---|---|
| High Level I/O | *Streams* | |
| Low Level I/O | *File Descriptors* | What we covered in Lecture 4 |
| Syscall | *open(), read(), write(), close(), …*<br>*Open File Descriptions* | |
| File System | *Files/Directories/Indexes* | What we are covering… |
| I/O Driver | *Commands and Data Transfers* | What we previously covered… |
| | *Disks, Flash, Controllers, DMA* | |

# Recall: From Storage to File Systems

| | | |
|---|---|---|
| I/O API and syscalls | Variable-Size Buffer | *Memory Address* |
| File System | Block | *Logical Index, Typically 4 KB* |

**Hardware Devices**

**HDD**
- Sector(s)
- Physical Index, 512B or 4KB

**SSD**
- Flash Trans. Layer
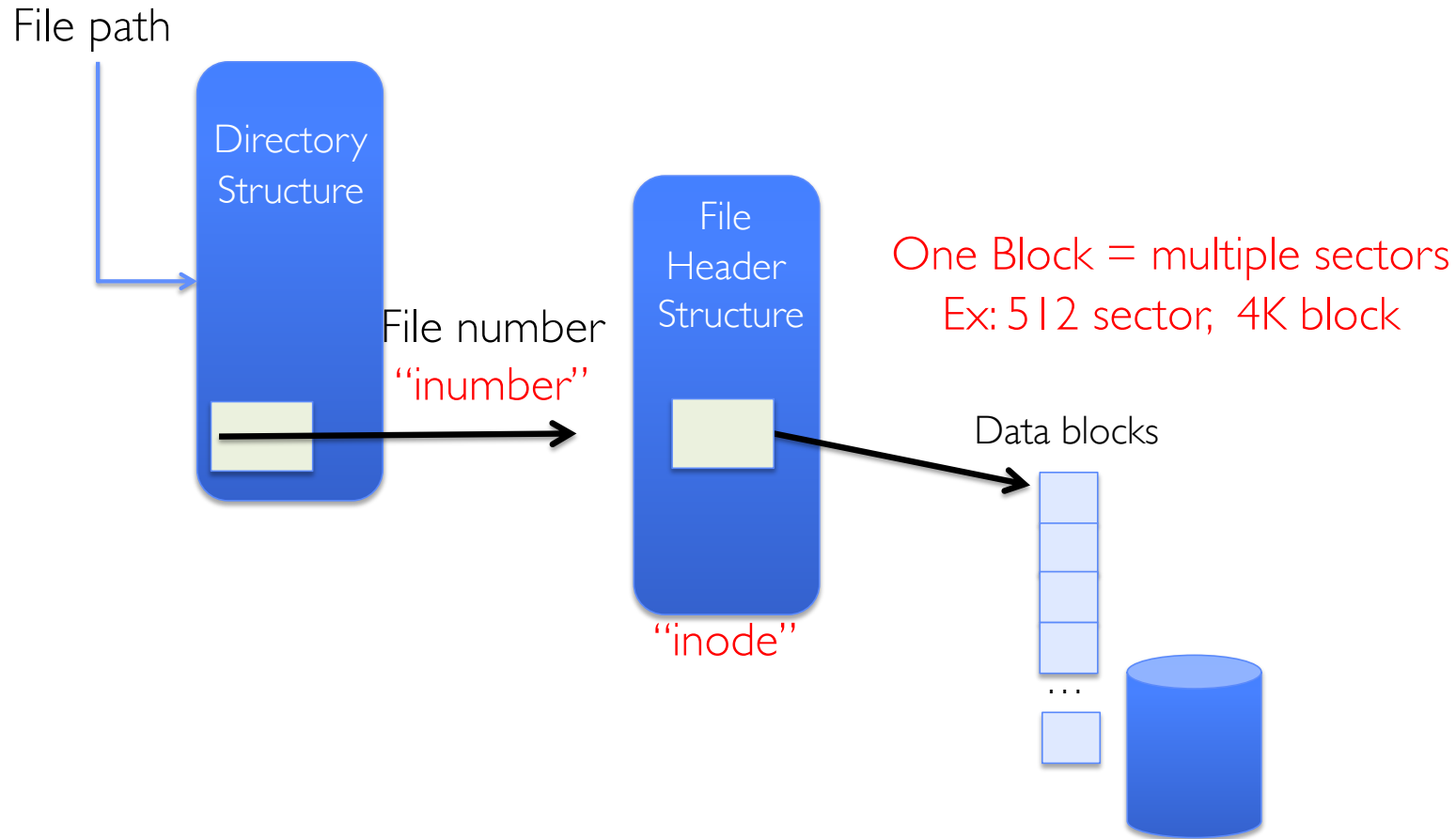- Phys. Block — *Phys Index., 4KB*
- Erasure Page
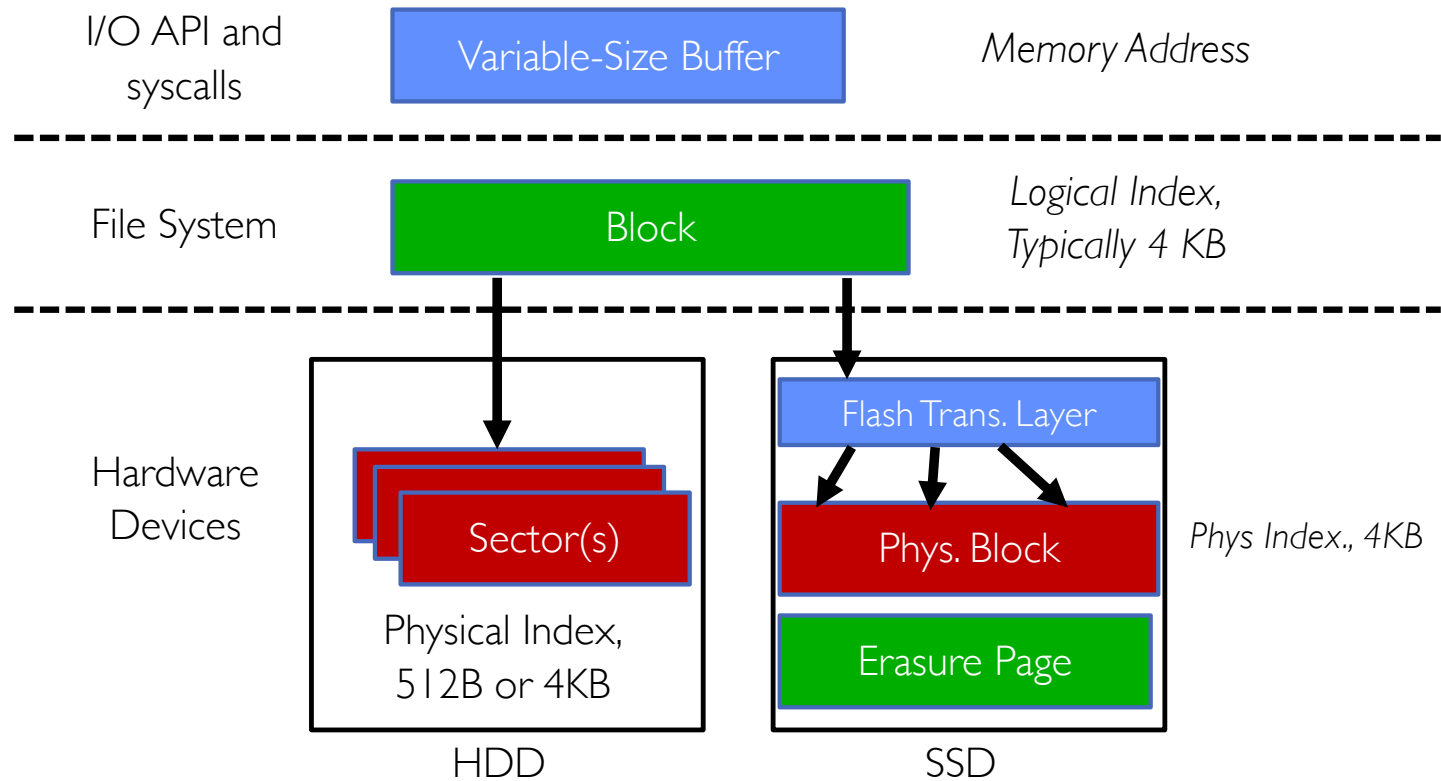
# Recall: Building a File System

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

- Classic OS situation: Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with:
  - Naming: Find file by name, not block numbers
  - Organize file names with directories
  - Organization: Map files to blocks
  - Protection: Enforce access restrictions
  - Reliability: Keep files intact despite crashes, hardware failures, etc.
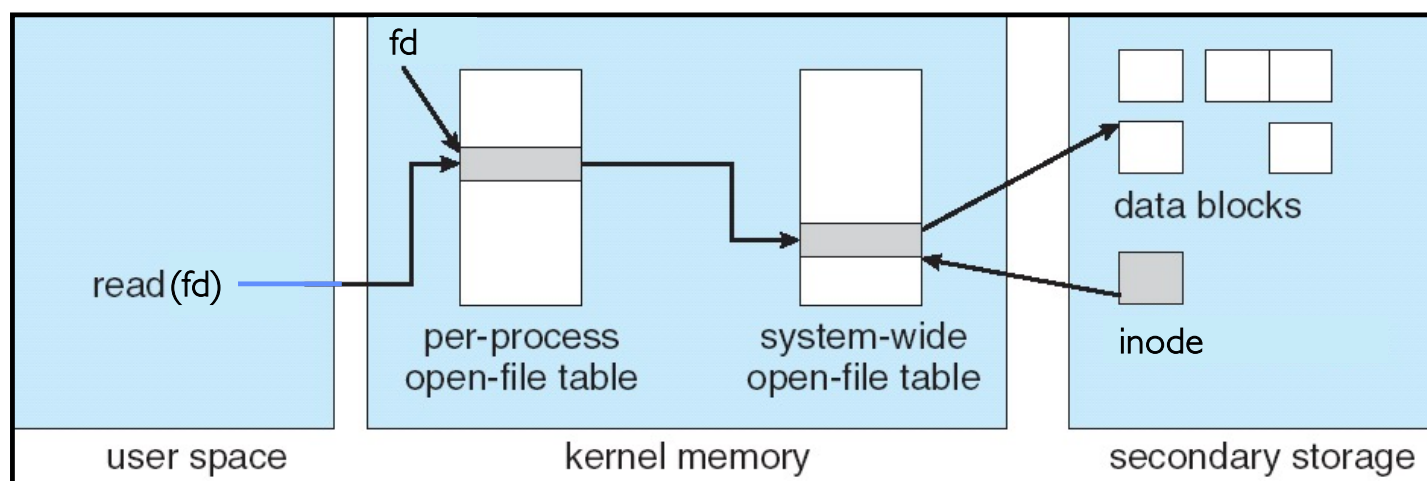
# Recall: User vs. System View of a File

- 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!

- System's view (inside OS):
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size ≥ sector size; in UNIX, block size is 4KB

# Recall: Components of a File System

File path

Directory
Structure

File number
"inumber"

File
Header
Structure

One Block = multiple sectors
Ex: 512 sector, 4K block
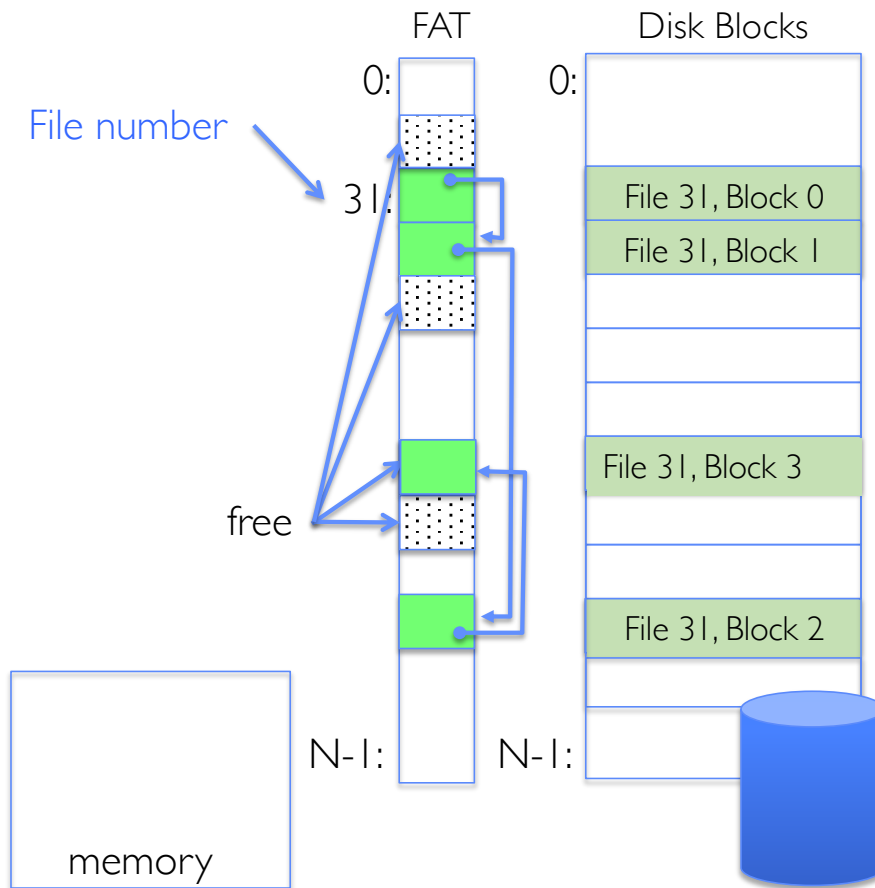
Data blocks

"inode"
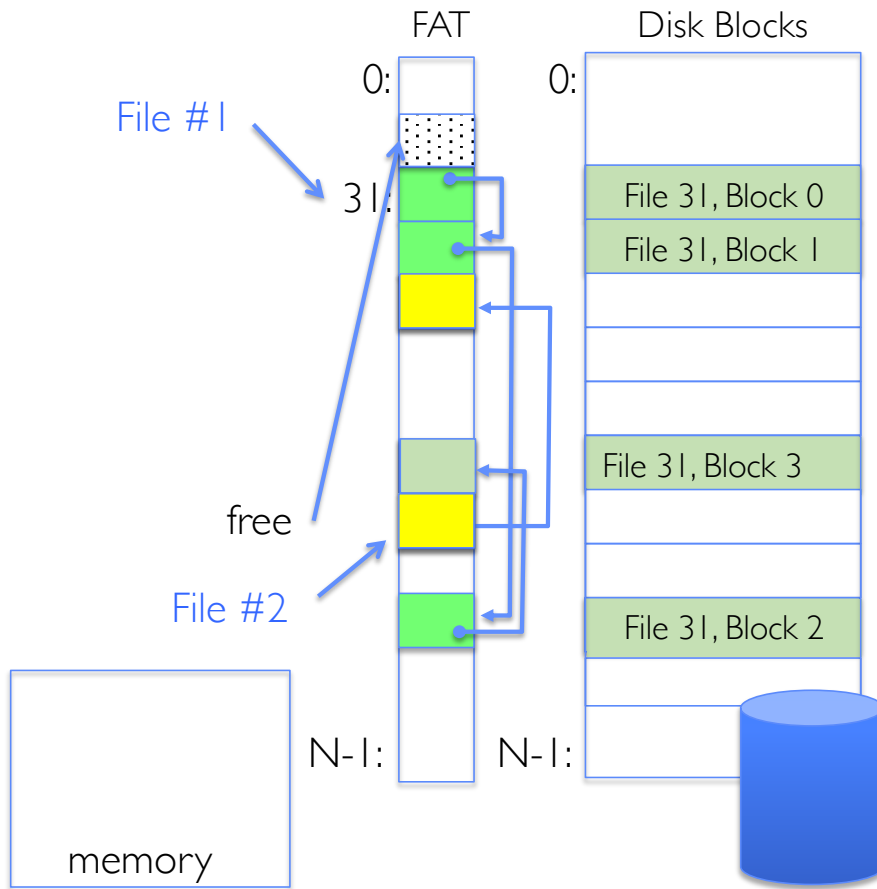
...

# Recall: In-Memory File System Structures



- Open syscall: find inode on disk from pathname (traversing directories)
  - Create "in-memory inode" in system-wide open file table
  - One entry in this table no matter how many instances of the file are open
- Read/write syscalls look up in-memory inode using the file handle

# Recall: FAT (File Allocation Table) Properties

FAT

Disk Blocks

File number

0:

0:

31:

File 31, Block 0

File 31, Block 1

File 31, Block 3

free

File 31, Block 2

N-1:

N-1:

memory

- File is collection of disk blocks ("clusters")
- FAT is array of integers mapped 1-1 with disk blocks
  - Each integer is either:
    - » Pointer to next block in file; or
    - » End of file flag; or
    - » Free block flag
- File Number is index of root of block list for the file
  - Follow list to get block #
  - Directory must map name to block number at start of file
- But: Where is FAT stored?
  - Beginning of disk, before the data blocks
  - Usually 2 copies (to handle errors)

# Recall: FAT Discussion

FAT | Disk Blocks

File #1

0:

31:

free

File #2

N-1:

memory

0:

File 31, Block 0

File 31, Block 1

File 31, Block 3

File 31, Block 2

N-1:

Suppose you start with the file number:

- Time to find block?
- Block layout for file?
- Sequential access?
- Random access?
- Fragmentation?
- Small files?
- Big files?

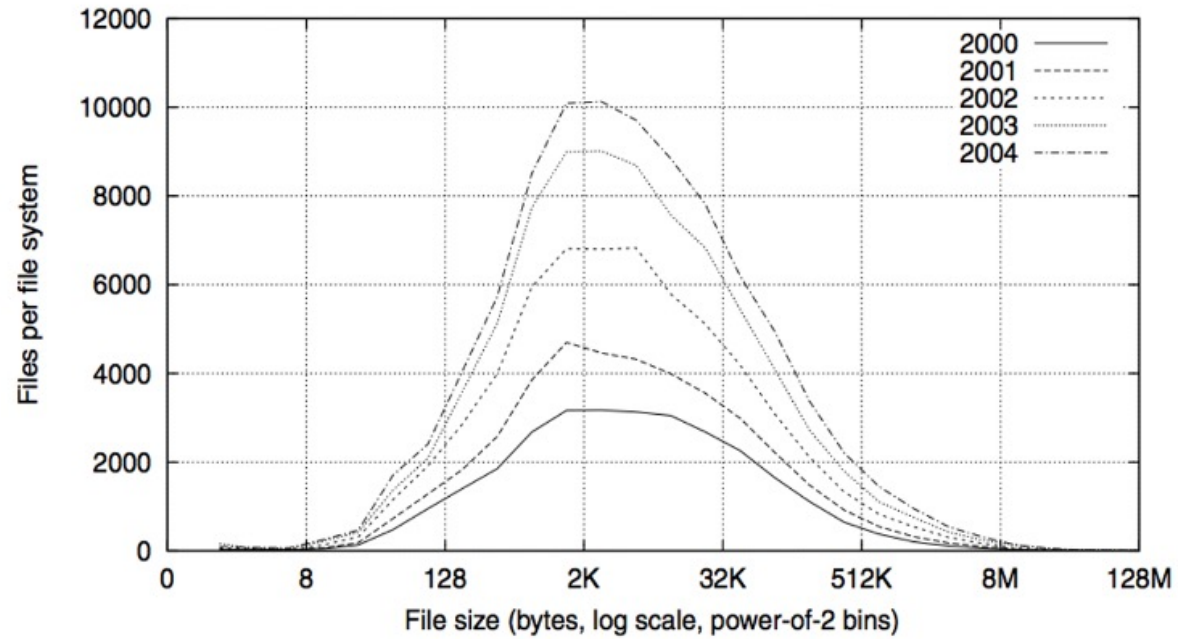# Recall: Observation #1: Most Files Are Small



Fig. 2. Histograms of files by size.

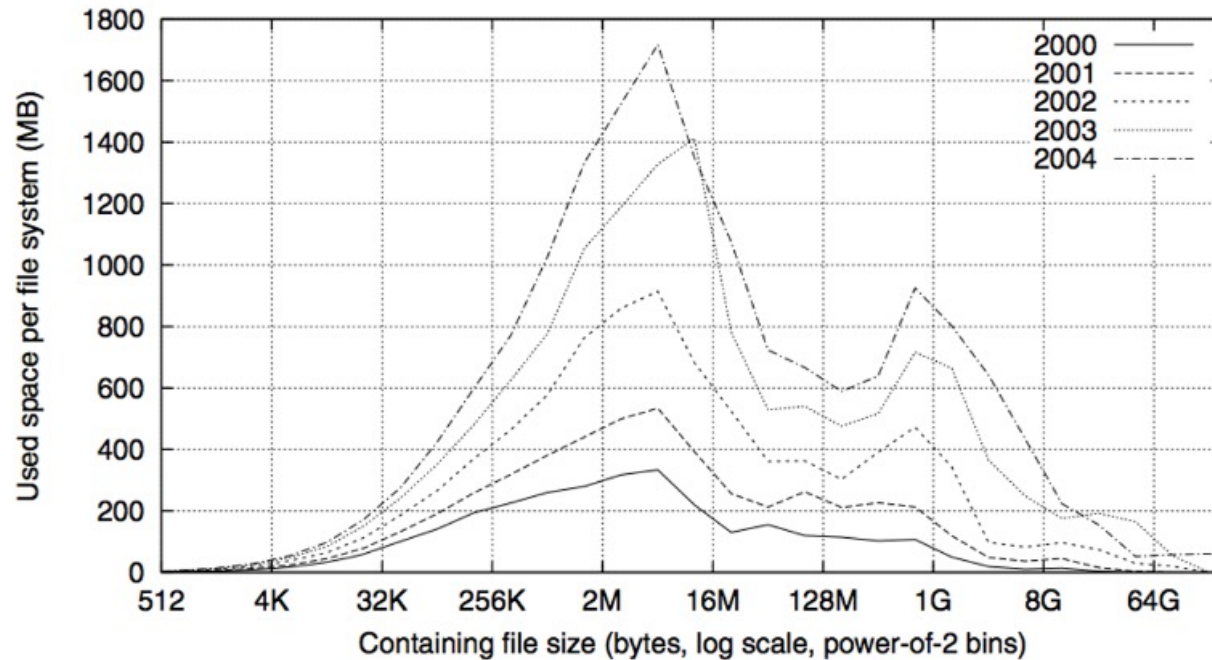# Recall: Observation #2: Most Bytes are in Large Files



Fig. 4. Histograms of bytes by containing file size.

# Recall: Multilevel Indexed Files (Original 4.1 BSD)

- Sample file in multilevel indexed format:
  - 10 direct ptrs, 1K blocks
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
  - Pros:    Simple (more or less)
            Files can easily expand (up to a point)
            Small files particularly cheap and easy
  - Cons:   Lots of seeks
            Very large files must read many indirect block
            (Four I/Os per block!)
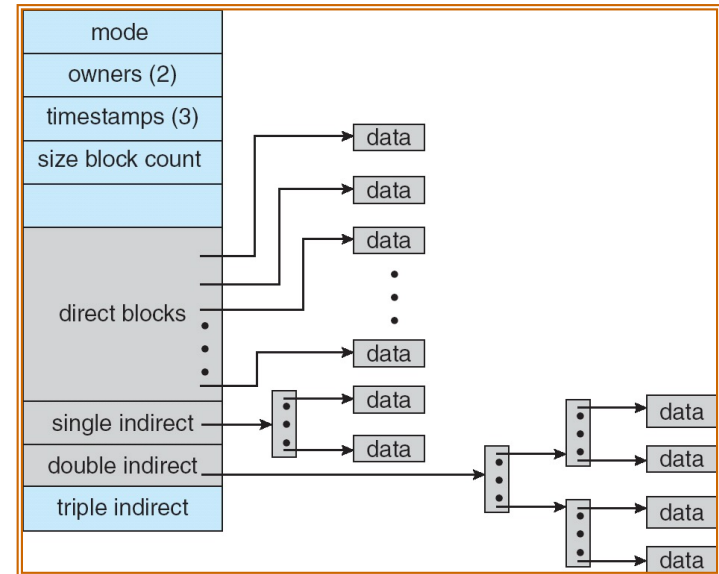
# Recall: Fast File System (BSD 4.2, 1984)

- Same inode structure as in BSD 4.1
  - same file header and triply indirect blocks like we just studied
  - Some changes to block sizes from 1024 $\Rightarrow$ 4096 bytes for performance
- Paper on FFS: "A Fast File System for UNIX"
  - Marshall McKusick, William Joy, Samuel Leffler and Robert Fabry
  - Off the "resources" page of course website – Take a look!

- Optimization for Performance and Reliability:
  - Distribute inodes among different tracks to be closer to data
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned later)

# Recall: UNIX 4.2 BSD FFS

- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata and data
  - No defragmentation necessary!

- Cons
  - Inefficient for tiny files (a 1-byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on disk
  - Need to reserve 10-20% of free space to prevent fragmentation
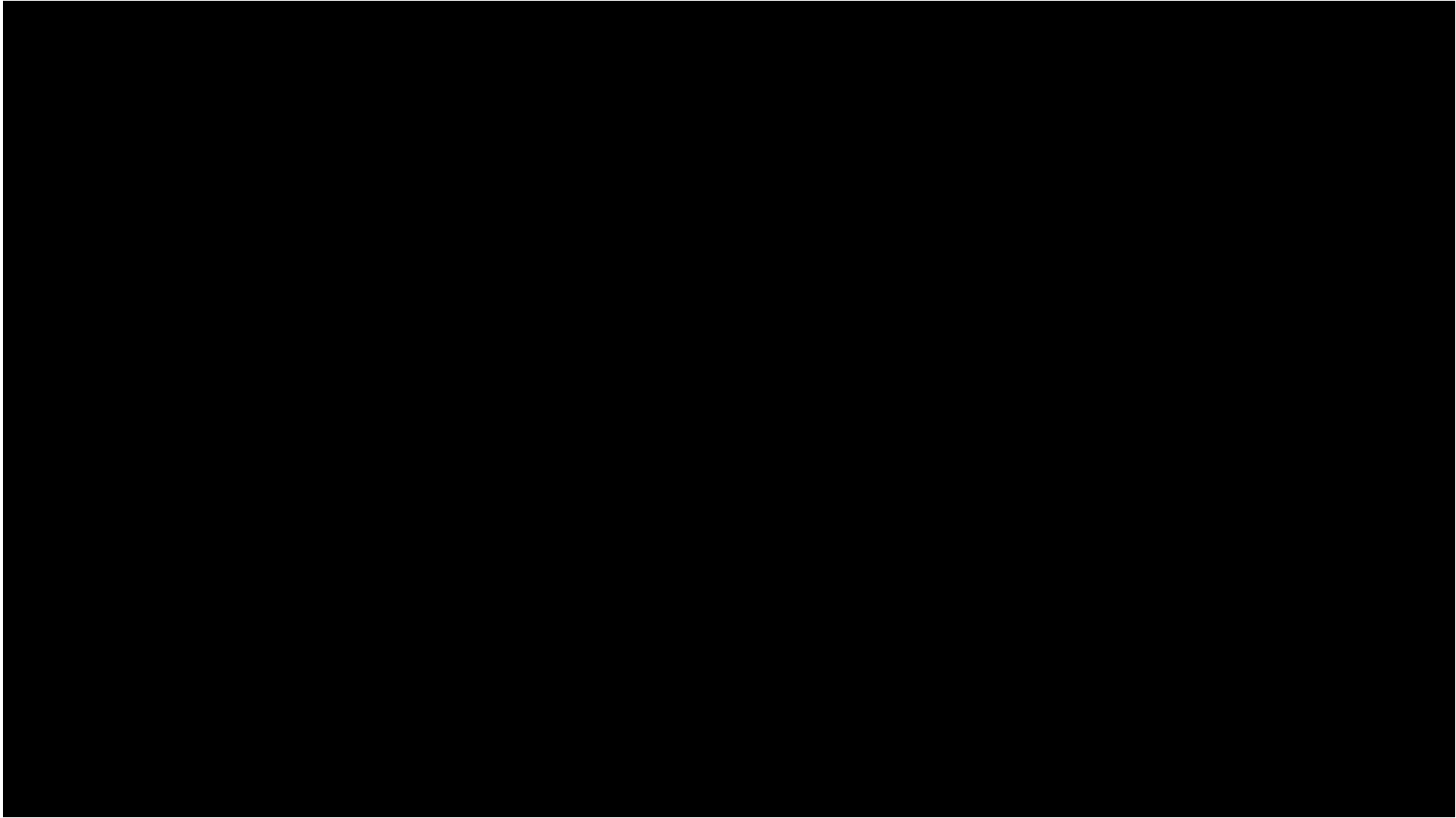
# Administrivia

- Make sure to fill out post midterm survey
    - Let us know how we are doing or what we could improve
- If you have any group issues going on, make sure you:
    - Make sure that your TA understands what is happing
    - Make sure that you reflect these issues on your group evaluations

# Recall: CS 162 Collaboration Policy

Explaining a concept to someone in another group

Discussing algorithms/testing strategies with other groups

Discussing debugging approaches with other groups

Searching online for generic algorithms (e.g., hash table)

Sharing OR verbally discussing code/test cases with another group
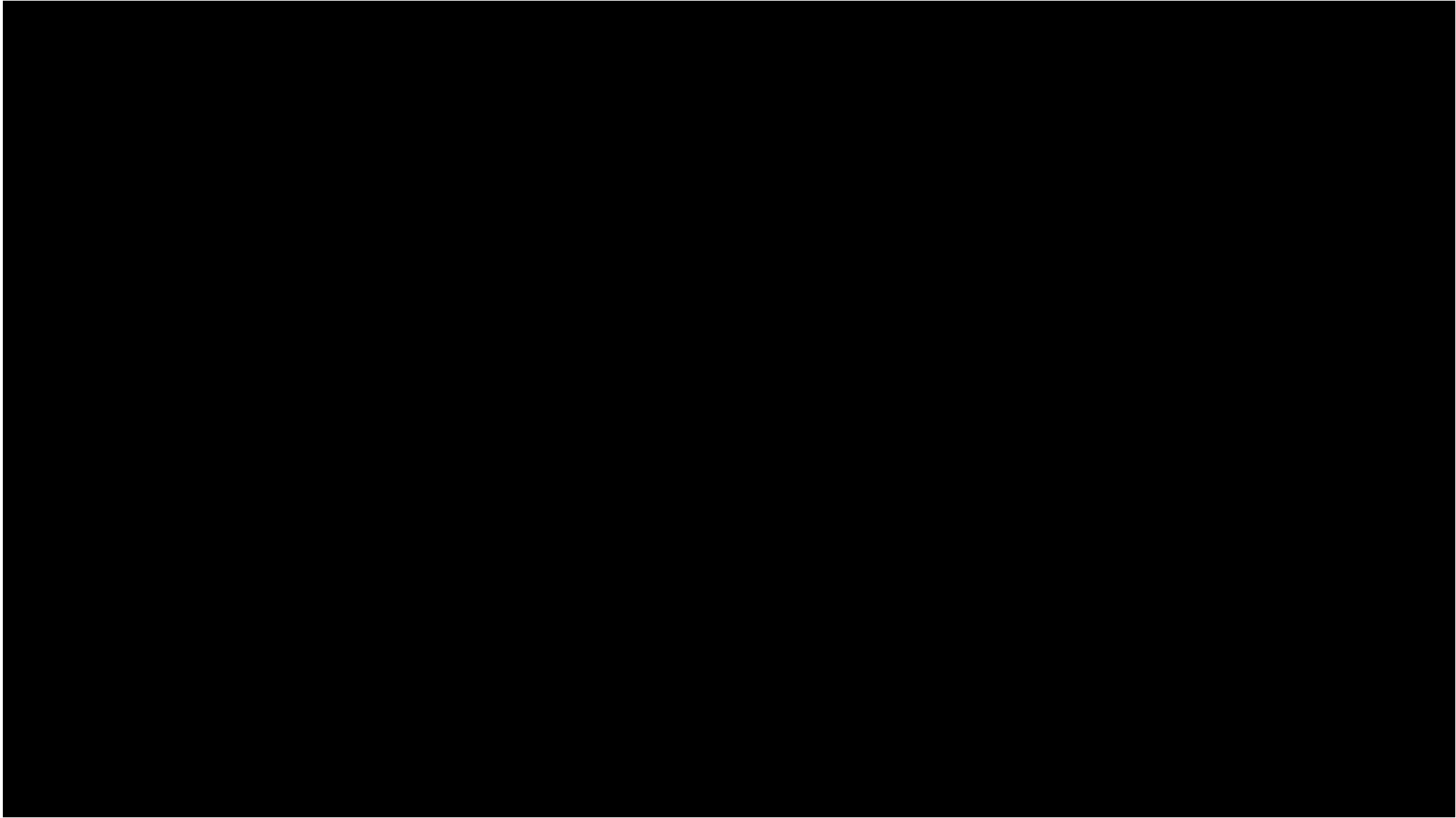(or a teammate for HW)

Copying OR reading another group's code or test cases

Copying OR reading online code or test cases from prior years

Helping someone in another group to debug their code

- We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders

- Don't put a friend in a bad position by asking for help that they shouldn't give!

# Linux Example: Ext2/3 Disk Layout
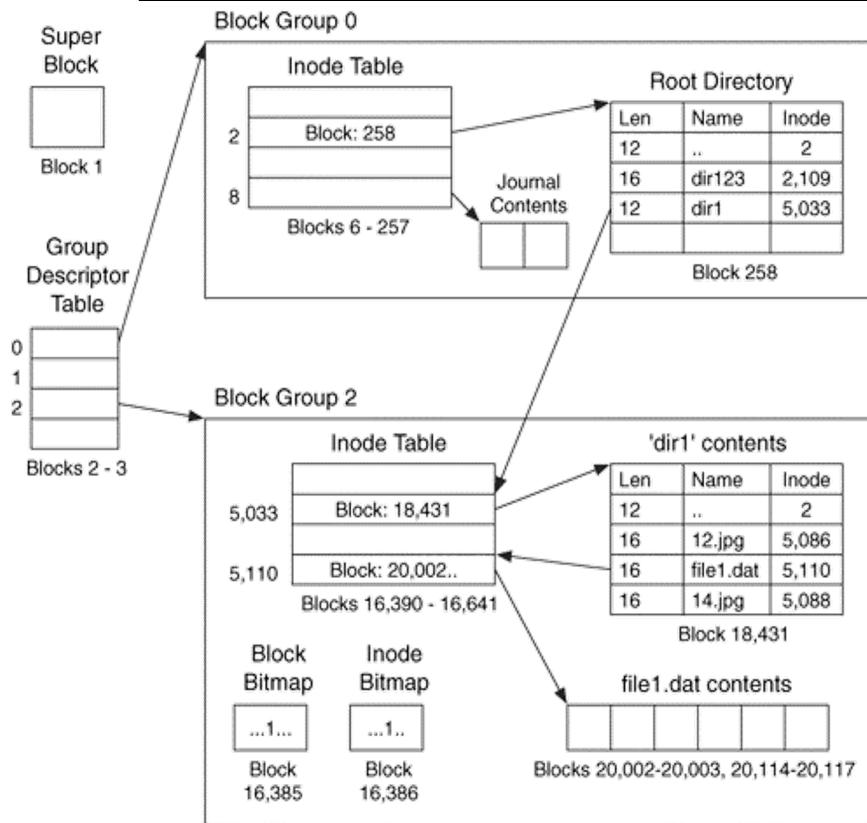


- Disk divided into block groups
  - Provides locality
  - Each group has two block-sized bitmaps (free blocks/inodes)
  - Block sizes settable at format time: 1K, 2K, 4K, 8K…
- Actual inode structure similar to 4.2 BSD
  - With 12 direct pointers instead of 10
- Ext3: Ext2 with Journaling
  - Several degrees of protection with comparable overhead
  - We will talk about Journaling later

Example: create a `file1.dat` under /dir1/ in Ext3

# Recall: Directory Abstraction

- Directories are specialized files
  - Contents: **List of pairs  <file name, file number>**
- System calls to access directories
  - **open** / **creat** traverse the structure
  - **mkdir** /**rmdir** add/remove entries
  - **link** / **unlink** (**rm**)
- libc support
  - DIR * opendir (const char *dirname)
  - struct dirent * readdir (DIR *dirstream)
  - int readdir_r (DIR *dirstream,
                  struct dirent *entry,
                  struct dirent **result)

/usr

/usr/lib

/usr/lib4.3

/usr/lib4.3/foo
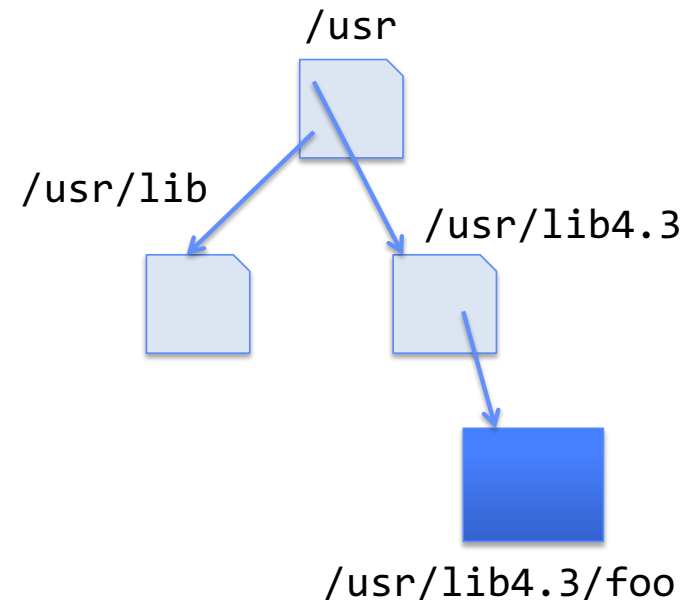
# Hard Links

- Hard link
  - Mapping from name to **file number** in the directory structure
  - First hard link to a file is made when file created
  - Create extra hard links to a file with the `link()` system call
  - Remove links with `unlink()` system call
- When can file contents be deleted?
  - When there are no more hard links to the file
  - Inode maintains **reference count** for this purpose

/usr

/usr/lib

/usr/lib4.3

/usr/lib4.3/foo
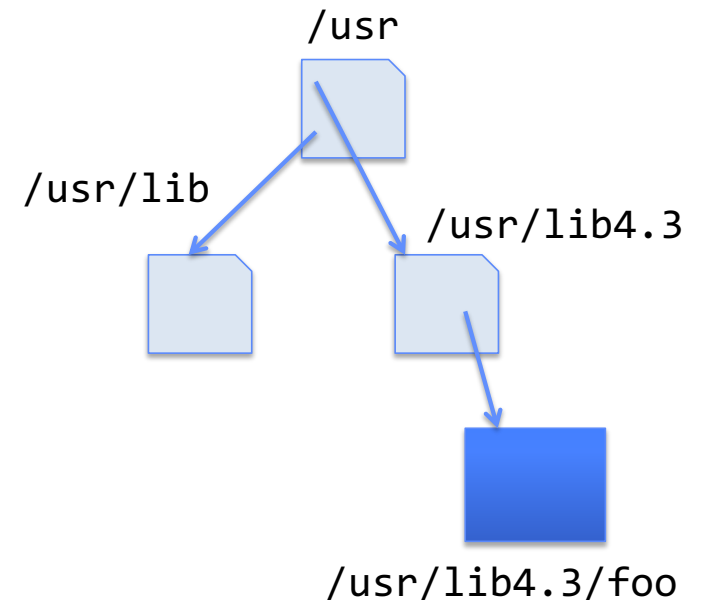
# Soft Links (Symbolic Links)

- Soft link or Symbolic Link or Shortcut
  - Directory entry contains the path and name of the file
  - Map one name to another name

- Contrast these two different types of directory entries:
  - Normal directory entry: **<file name, file #>**
  - Symbolic link: **<file name, destination file name>**

- OS looks up destination file name **each time** program accesses source file name
  - Lookup can fail (error result from **open**)

- Unix: Create soft links with **symlink** syscall

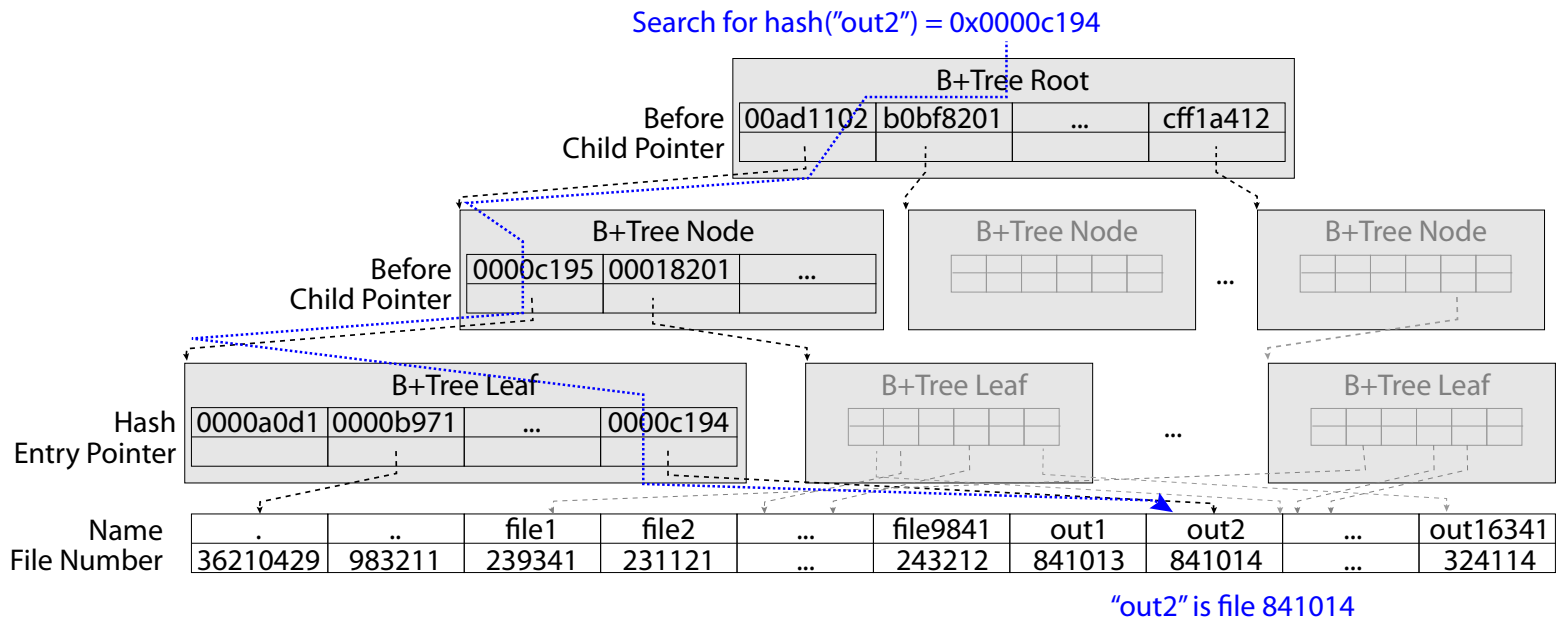# Directory Traversal



inode

block 49358

"home":8086

block 12132

"stuff.txt":9909

block 7756

"cs162":732

Blocks of
stuff.txt

2   9099   "home":8086

732

8086   "cs162":732   "stuff.txt":9909

memory
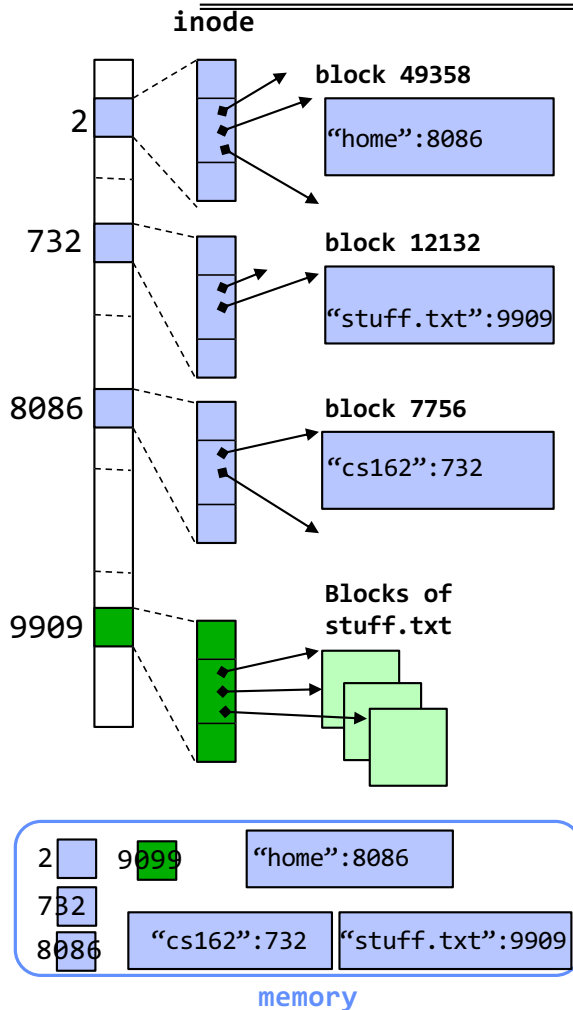
- What happens when we open **/home/cs162/stuff.txt**?
- "**/**" - inumber for root inode configured into kernel, say **2**
    - Read inode **2** from its position in inode array on disk
    - Extract the direct and indirect block pointers
    - Determine block that holds root directory (say block **49358**)
    - Read that block, scan it for "**home**" to get inumber for this directory (say **8086**)
- Read inode **8086** for **/home**, extract its blocks, read block (say **7756**), scan it for "**cs162**" to get its inumber (say **732**)
- Read inode **732** for **/home/cs162**, extract its blocks, read block (say **12132**), scan it for "**stuff.txt**" to get its inumber, say **9909**
- Read inode **9909** for **/home/cs162/stuff.txt**
- Set up file description to refer to this inode so reads / write can access the data blocks referenced by its direct and indirect pointers
- Check permissions on each directory's inode AND the final inode

# Large Directories: B-Trees (dirhash)

in FreeBSD, NetBSD, OpenBSD

Search for hash("out2") = 0x0000c194

| B+Tree Root | | | |
|---|---|---|---|
| 00ad1102 | b0bf8201 | ... | cff1a412 |

Before Child Pointer

| B+Tree Node | | |
|---|---|---|
| 0000c195 | 00018201 | ... |

Before Child Pointer

B+Tree Node

B+Tree Node

| B+Tree Leaf | | | |
|---|---|---|---|
| 0000a0d1 | 0000b971 | ... | 0000c194 |

Hash Entry Pointer

B+Tree Leaf

B+Tree Leaf

| Name | . | .. | file1 | file2 | ... | file9841 | out1 | out2 | ... | out16341 |
|---|---|---|---|---|---|---|---|---|---|---|
| File Number | 36210429 | 983211 | 239341 | 231121 | ... | 243212 | 841013 | 841014 | ... | 324114 |

"out2" is file 841014

# CASE STUDY:
# WINDOWS NTFS

# New Technology File System (NTFS)

- Default on modern Windows systems
- Variable length extents
  - Rather than fixed blocks
- Instead of FAT or inode array: **Master File Table**
  - Like a database, with maximum of 1 KB size for each table entry
  - Everything (almost) is a sequence of <attribute:value> pairs
    - » Including metadata and data
- Each entry in MFT contains metadata and:
  - File's data directly (for small files)
  - A list of *extents* (start block, size) for file's data
  - For big files: pointers to other MFT entries with *more* extent lists

# NTFS
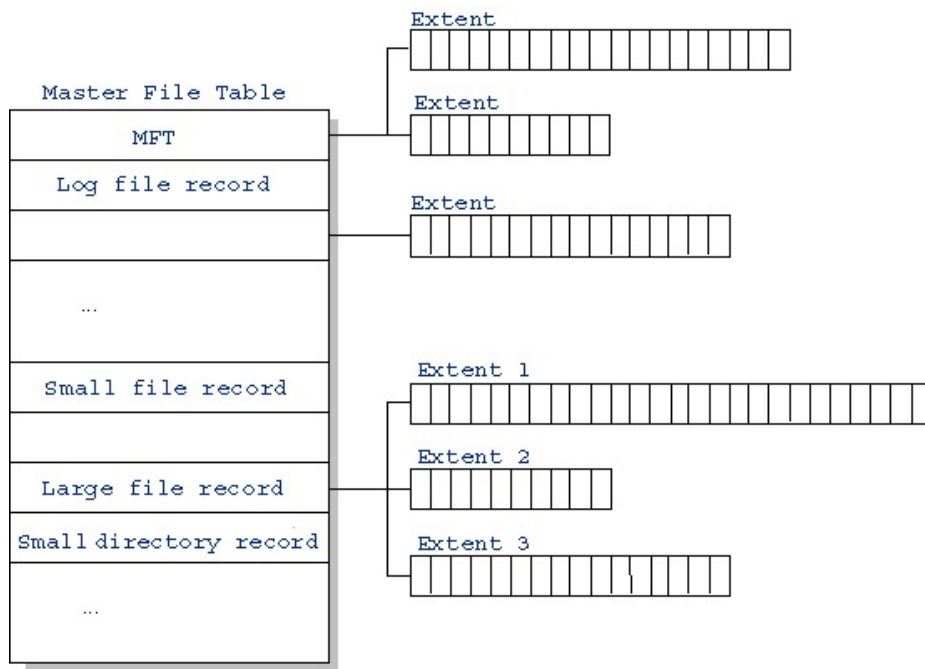
- Master File Table
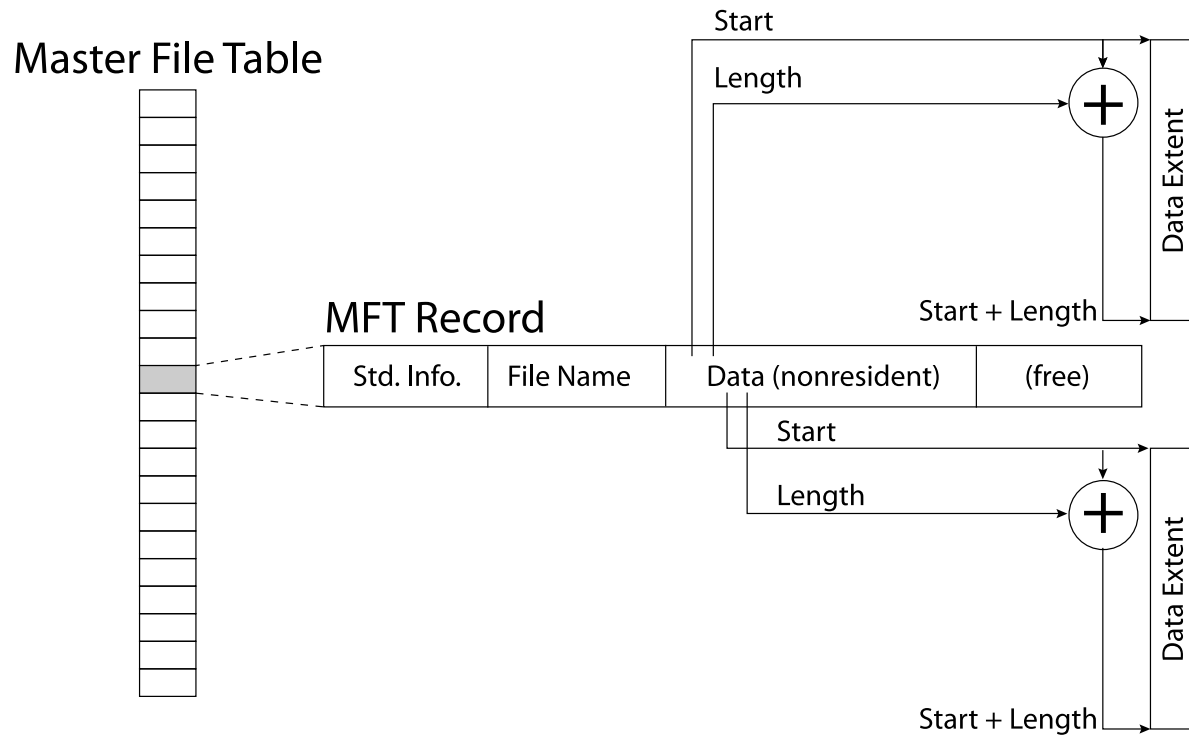  - Database with flexible 1KB entries for metadata/data
  - Variable-sized attribute records (data or metadata)
  - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
  - Block pointers cover runs of blocks
  - Similar approach in Linux (ext4)
  - File create can provide hint as to size of file
- Journaling for reliability
  - Discussed later

# NTFS Small File: Data stored with Metadata

## Master File Table

Create time, modify time, access time,
Owner id, security specifier, flags (RO, hidden, sys)

data attribute

### MFT Record (small file)

| Std. Info. | File Name | Data (resident) | (free) |
|------------|-----------|-----------------|--------|

Attribute list

# NTFS Medium File: Extents for File Data

# NTFS Large File: Pointers to Other MFT Records

Master File Table

MFT Record
(huge/badly-fragmented file)

| Std. Info. | Attr. List (nonresident) | ... |

...

... Extent with part of attribute list

Data (nonresident)

...

Data (nonresident)

...

Data (nonresident)

...

... Extent with part of attribute list

Data (nonresident)

...

Data (nonresident)

...

... Extent with part of attribute list

Data (nonresident)

...

Data (nonresident)

...
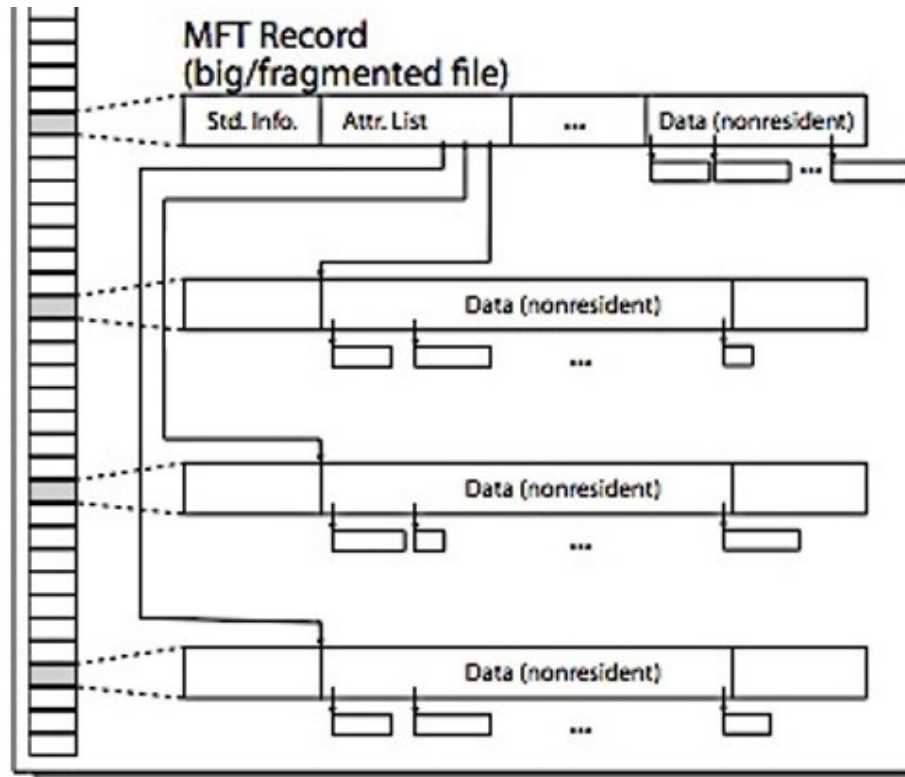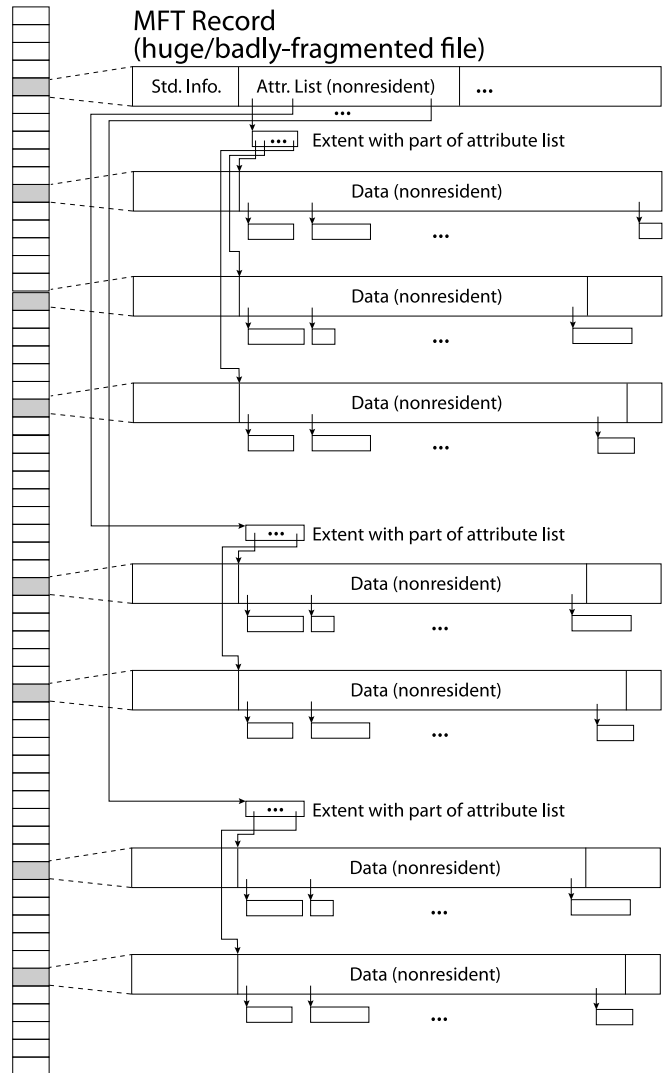
# NTFS Huge, Fragmented File: Many MFT Records

# NTFS Directories

- Directories implemented as B Trees – efficient for searching large directories

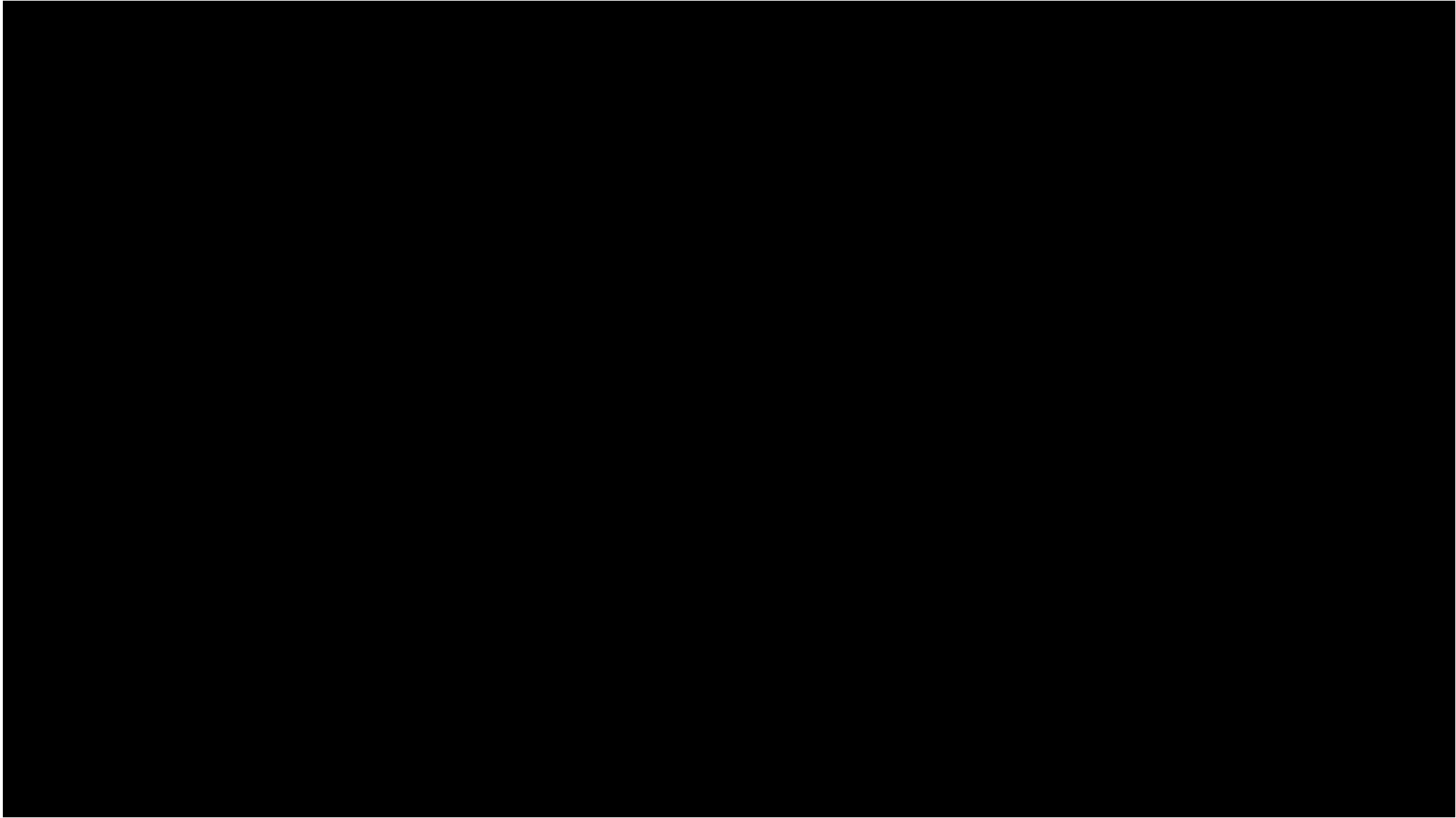- File's number identifies its entry in MFT

- MFT entry always has a file name attribute
  - Human readable name, file number of parent dir

- Hard link? Multiple file name attributes in MFT entry

# MEMORY MAPPED FILES

# Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to/from regions of a file
  - This involves multiple copies into caches in memory, plus system calls

- What if we could "map" the file directly into an empty region of our address space
  - Implicitly "page it in" when we read it
  - Write it and "eventually" page it out

- Executable files are treated this way when we `exec` the process!!

# Recall: Who Does What, When?



Process

virtual address

instruction

retry

page fault

exception

Operating System

Page Fault Handler

update PT entry

scheduler

MMU

page#

PT

frame#

offset

frame#

offset

physical address

load page from disk
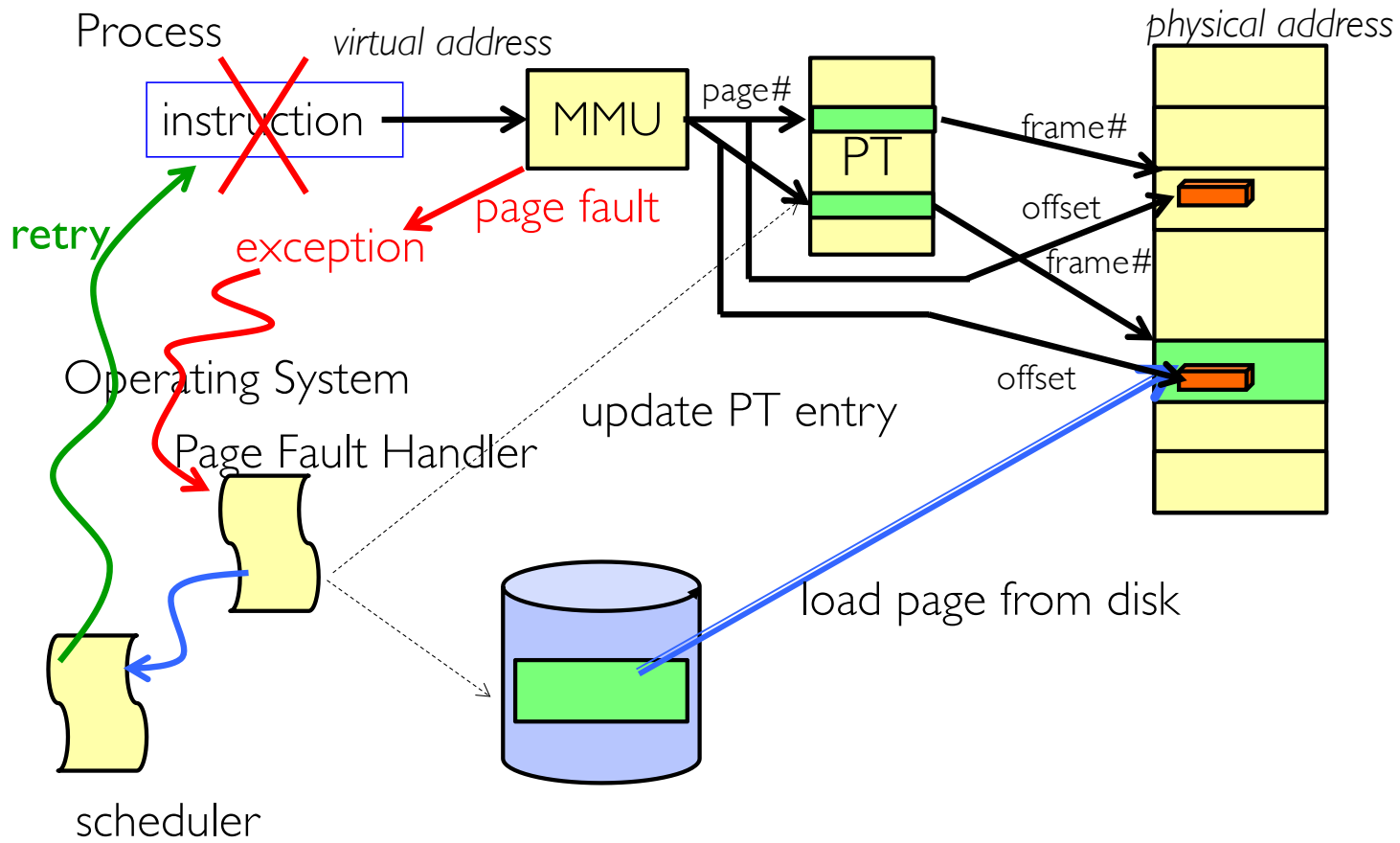
# Using Paging to `mmap()` Files



Process
virtual address
*physical address*

instruction → MMU → page# → PT → frame#
offset

retry

page fault

excep...

Read File contents from memory!

Operating System

Page Fault Handler

...ate PT entries for mapped region as "backed" by file

scheduler

File

`mmap()` file to region of VAS

# `mmap()` system call

```
MMAP(2)                   BSD System Calls Manual                   MMAP(2)

NAME
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- May map a specific region or let the system find one for you
  - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

# An mmap() Example

```
#include <sys/mman.h> /* also stdio.h, stdlib.h, string.h, fcntl.h, unistd.h */

int something = 162;

int main (int argc, char *argv[]) {
  int myfd;
  char *mfile;

  printf("Data  at: %16lx\n", (long
  printf("Heap at : %16lx\n", (long
  printf("Stack at: %16lx\n", (long

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_CR
  if (myfd < 0) { perror("open failed!

  /* map the file */
  mfile = mmap(0, 10000, PROT_READ|
  if (mfile == MAP_FAILED) {perror(

  printf("mmap at : %16lx\n", (long

  puts(mfile);
  strcpy(mfile+20,"Let's write over
  close(myfd);
  return 0;
}
```

```
$ ./mmap test
Data  at:        105d63058
Heap at :      7f8a33c04b70
Stack at:      7fff59e9db10
mmap at :        105d97000
This is line one
This is line two
This is line three
This is line four
```

```
$ cat test
This is line one
ThiLet's write over its line three
This is line four
```

Crooks & Joseph CS162 © UCB Spring 2021          Lec 20.40

# Sharing through Mapped Files

VAS 1

| VAS 1 |
|---|
| instructions |
| data |
| heap |
| |
| |
| stack |
| OS |

0x000…

0xFFF…

File

Memory

VAS 2

| VAS 2 |
|---|
| instructions |
| data |
| heap |
| |
| |
| stack |
| OS |

0x000…

0xFFF…
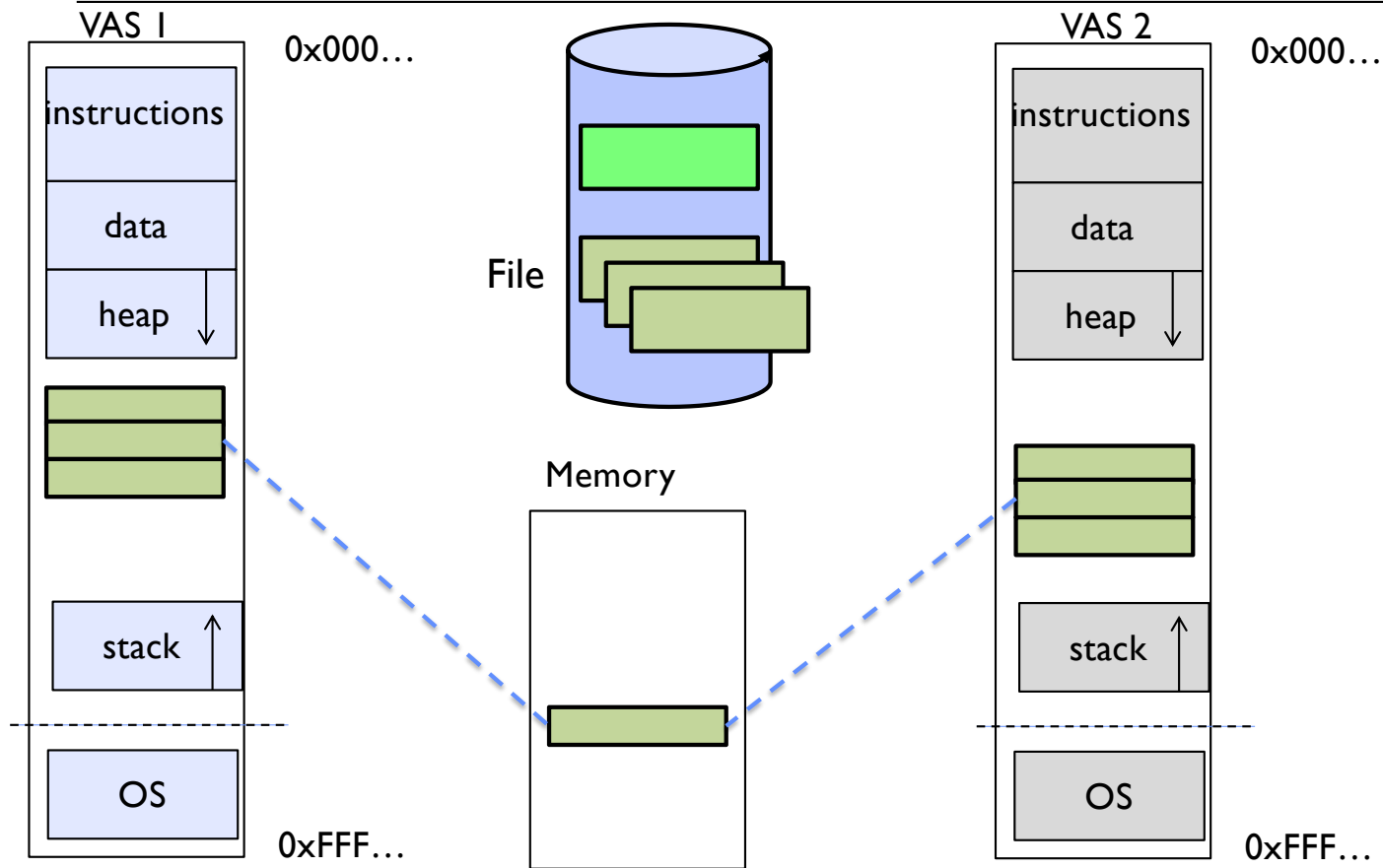
- Also: anonymous memory between parents and children
  - no file backing – just swap space

# THE BUFFER CACHE

# Buffer Cache

- Kernel *must* copy disk blocks to main memory to access their contents and write them back if modified
  - Could be data blocks, inodes, directory contents, etc.
  - Possibly dirty (modified and not written back)
- Key Idea: Exploit locality by caching disk data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (with modifications not on disk)

# File System Buffer Cache

Data blocks

Reading

PCB

iNodes

file
desc

Writing

Dir Data blocks

Free bitmap

Disk

Block Group 0

Block Group 1

Block Group 2

Memory
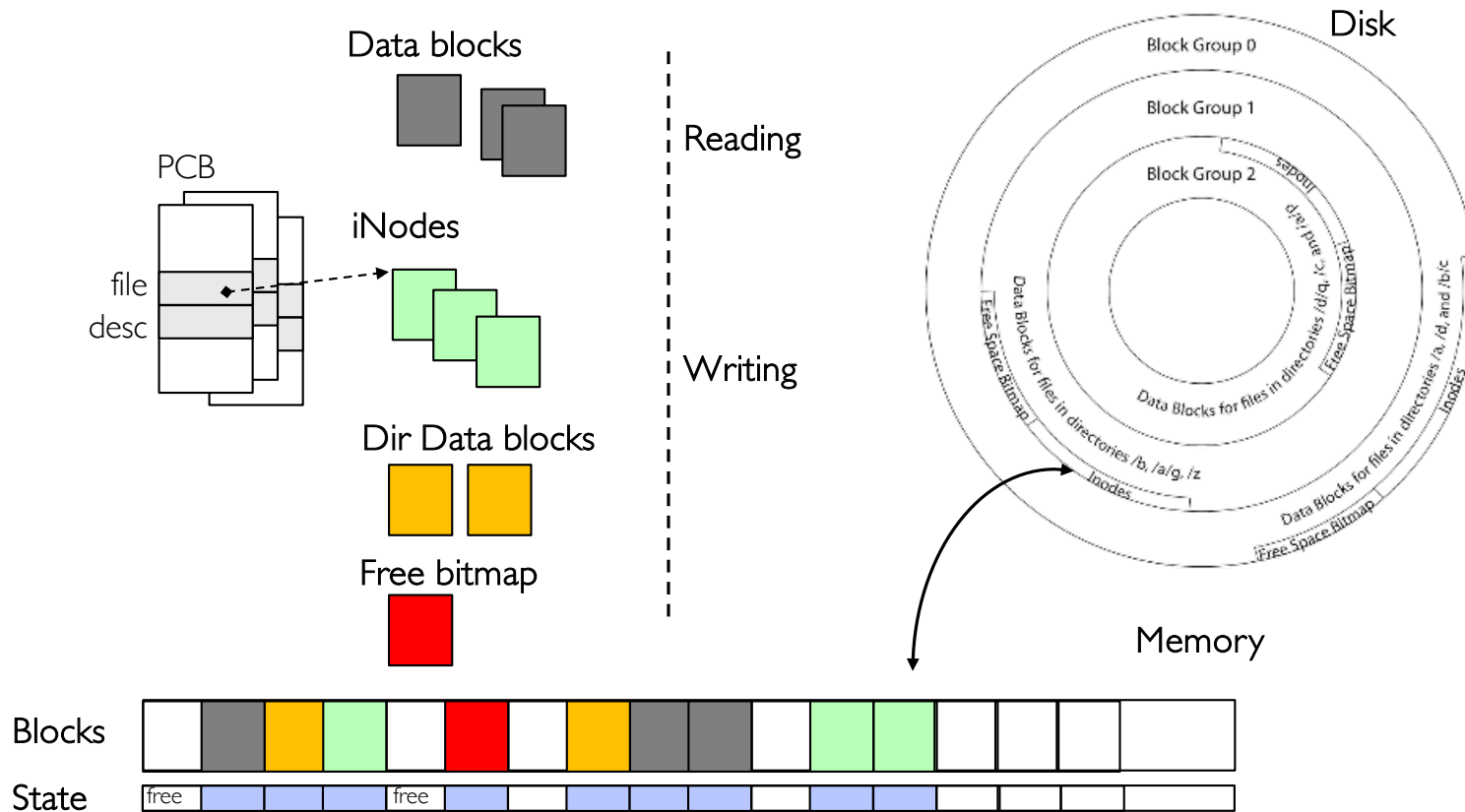
- OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap
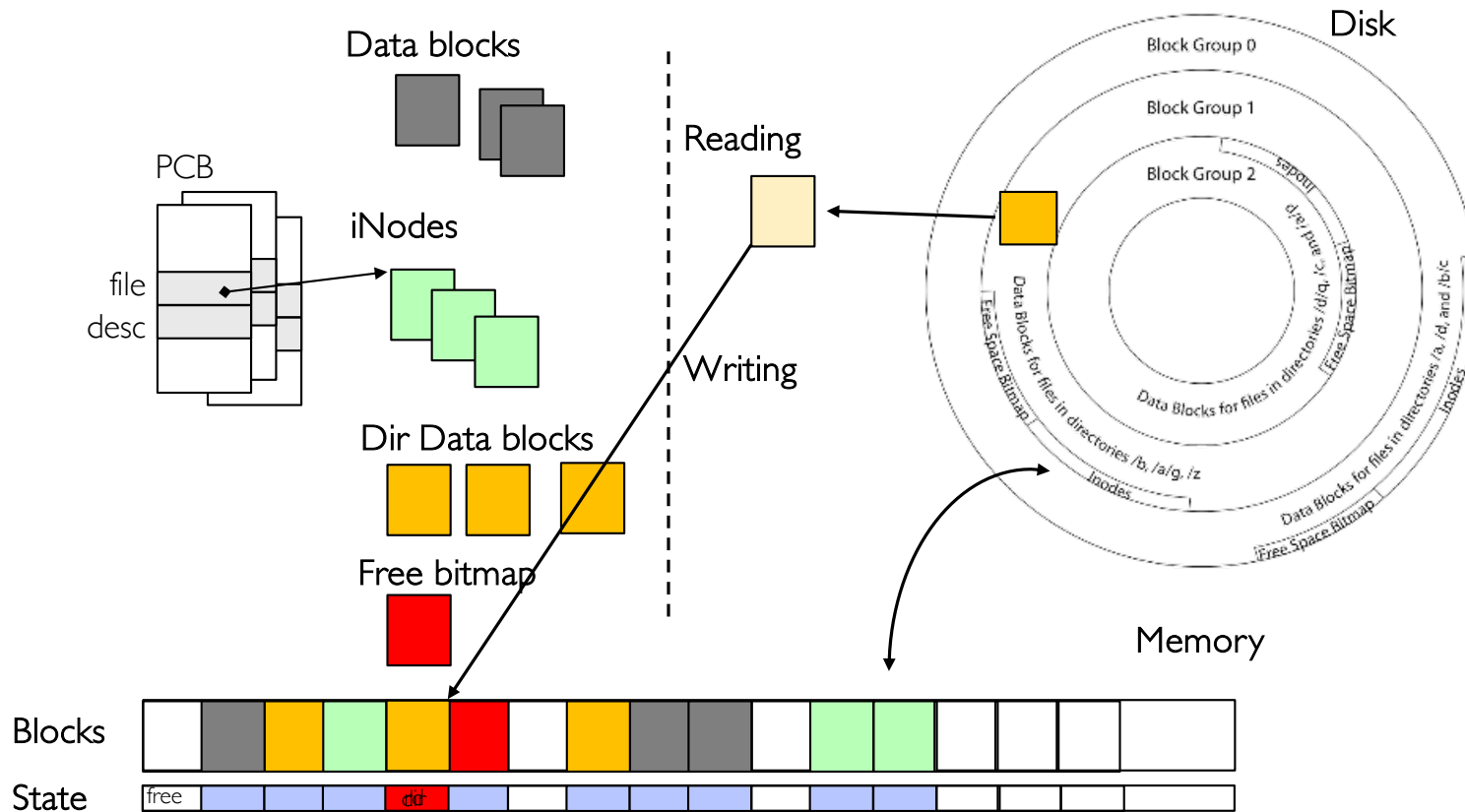
Blocks

State | free | | | free | | | | | | | | | | | | |

# File System Buffer Cache: Open

Data blocks

Reading

Writing

PCB

file desc

iNodes

Dir Data blocks

Free bitmap

Disk

Block Group 0

Block Group 1

Block Group 2

Memory

- Directory lookup repeat as needed:
  - load block of directory
  - search for map

Blocks

State

free    dd

# File System Buffer Cache: Open

**Data blocks**

**Reading**

Disk

Block Group 0

Block Group 1

Block Group 2

PCB

**iNodes**

file

desc

**Writing**

**Dir Data blocks**

<name>:inumber

**Free bitmap**

Memory

Blocks

State | free | | | dir | | | | | | | | rd | |
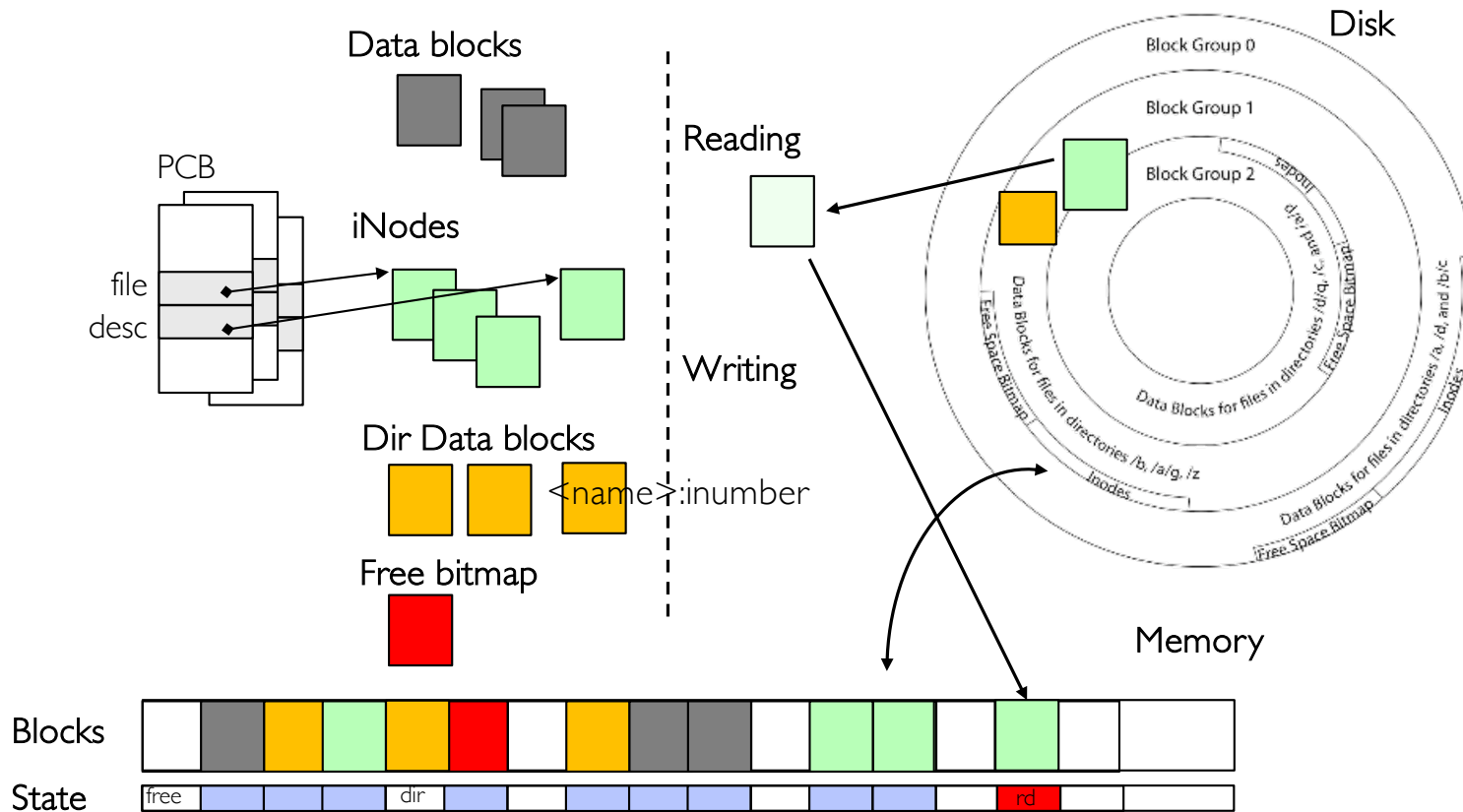
- Directory lookup repeat as needed:
  - load block of directory
  - search for map
- Create reference via open file descriptor

# File System Buffer Cache: Read?

**Data blocks**

**PCB**

file
desc

**iNodes**

Reading

Writing

**Dir Data blocks**

<name>:inumber

**Free bitmap**

**Disk**

Block Group 0

Block Group 1

Block Group 2

Memory
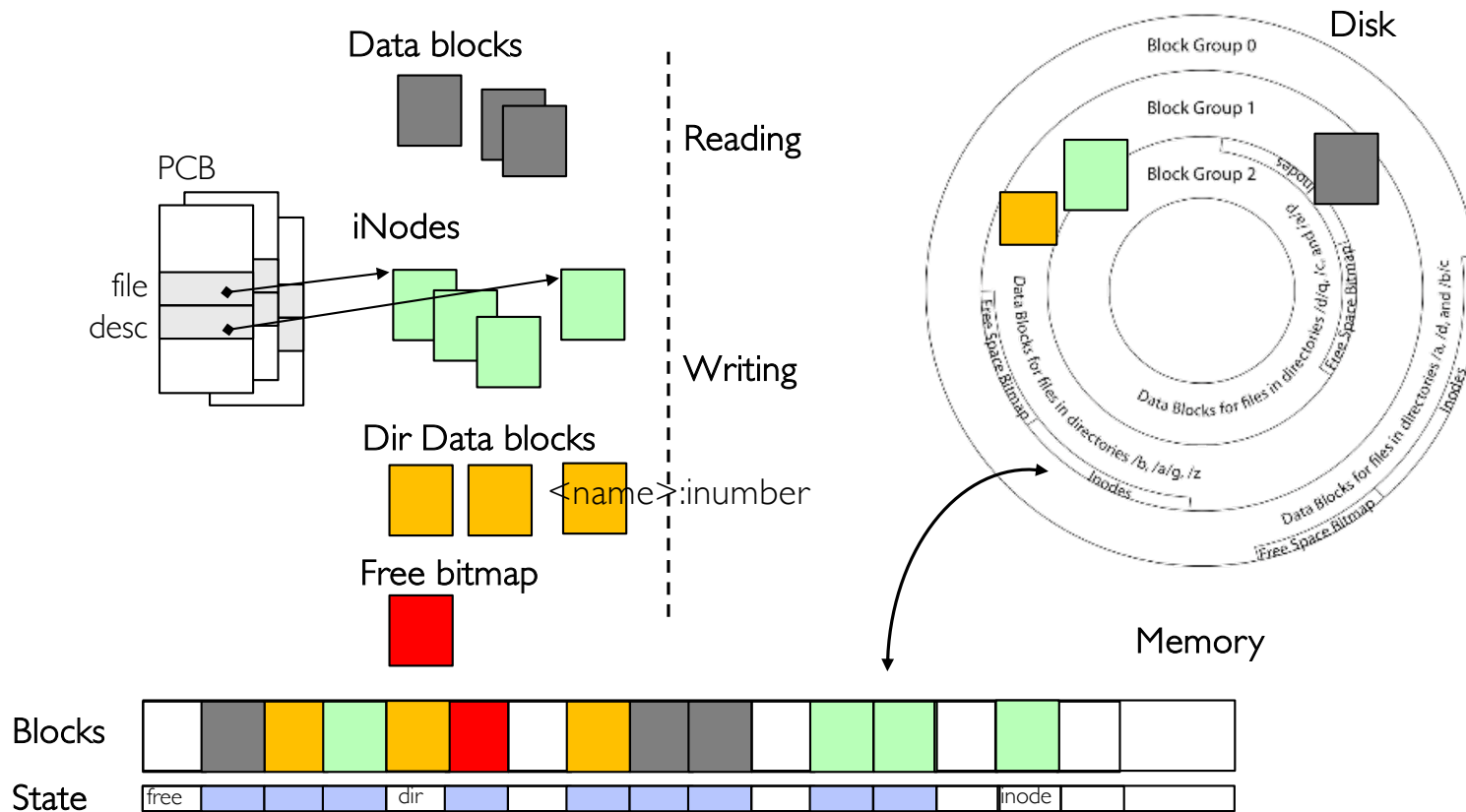
- Read Process
  - From inode, traverse index structure to find data block
  - load data block
  - copy all or part to read data buffer

| Blocks | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| State | free | | | dir | | | | | | | | inode | | |

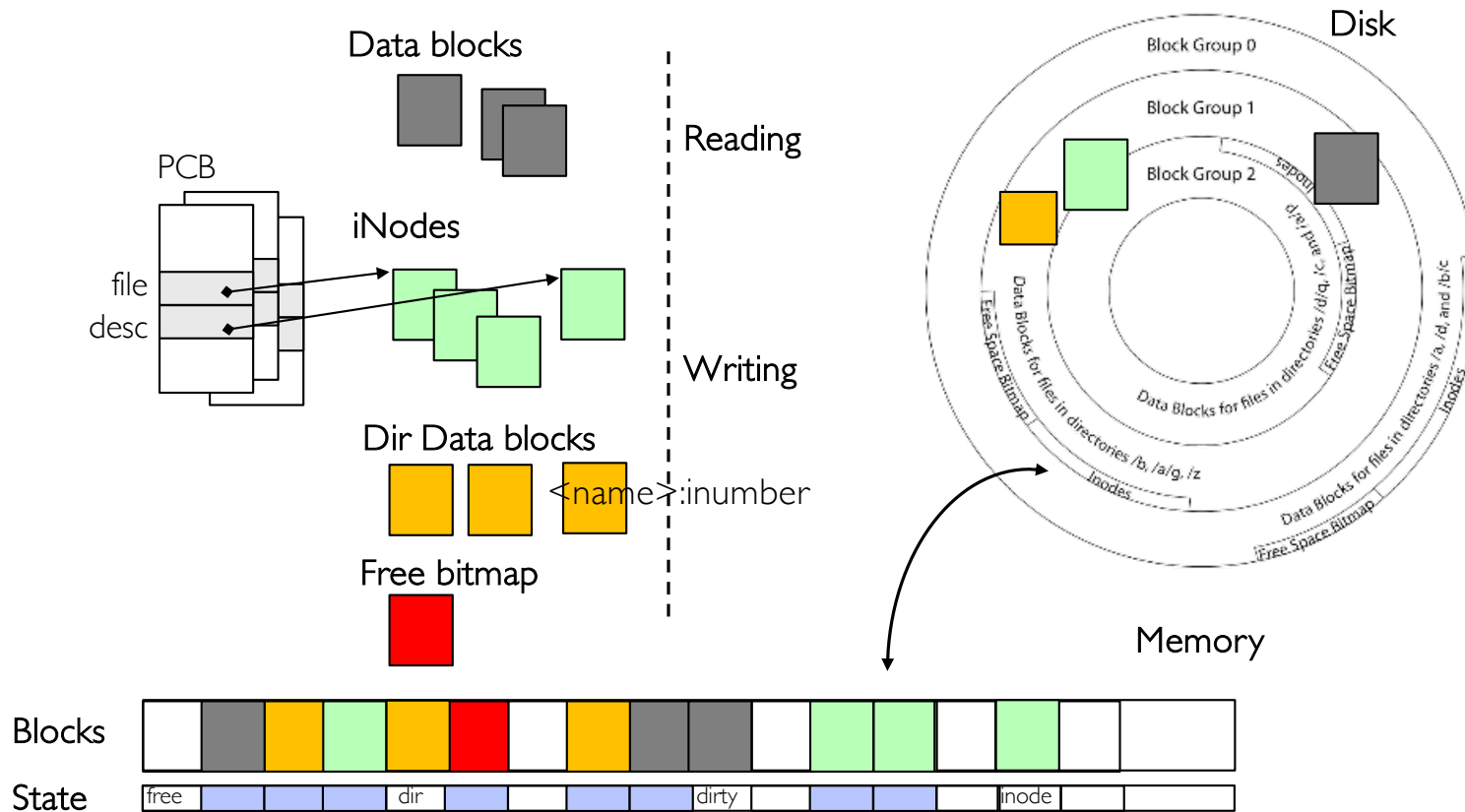# File System Buffer Cache: Write?

Data blocks

Reading

PCB

iNodes

file
desc

Writing

Dir Data blocks

<name>:inumber

Free bitmap

Disk

Block Group 0

Block Group 1

Block Group 2

Memory
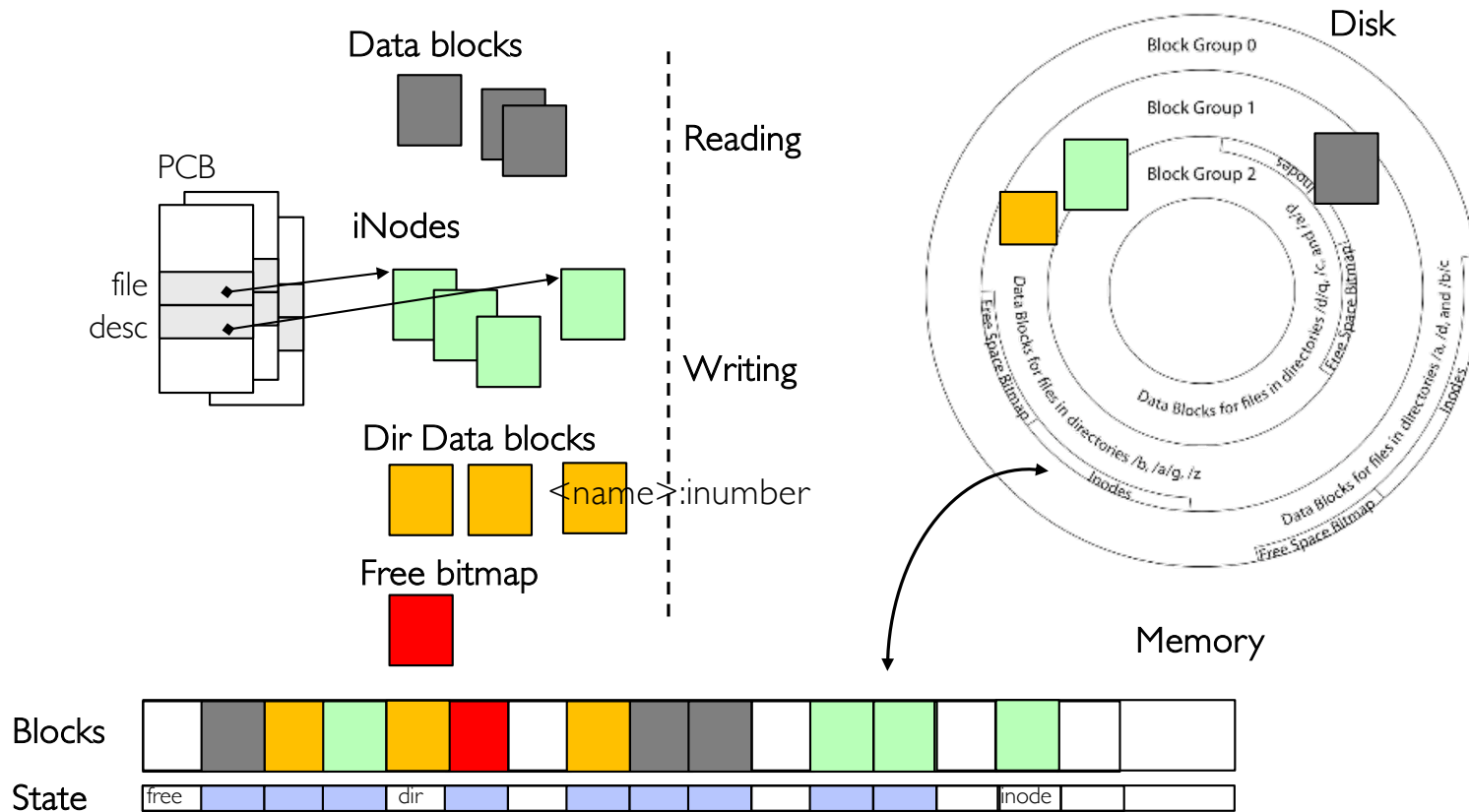
- Process similar to read, but may allocate new blocks (update free map), blocks need to be written back to disk; inode?

Blocks

State | free | | | | dir | | | | | | | | | | | inode | | |

# File System Buffer Cache: Eviction?



Data blocks

Reading

PCB

iNodes

file
desc

Writing

Dir Data blocks

<name>:inumber

Free bitmap

Disk

Block Group 0

Block Group 1

Block Group 2

Memory

Blocks

State    free          dir          dirty         inode

- Blocks being written back to disc go through a transient state

# Buffer Cache Discussion

- Implemented entirely in OS software
  - Unlike memory caches and TLB
- Blocks go through transitional states between free and in-use
  - Being read from disk, being written to disk
  - Other processes can run, etc.
- Blocks are used for a variety of purposes
  - inodes, data for dirs and files, freemap
  - OS maintains pointers into them
- Termination – e.g., process exit – open, read, write
- Replacement – what to do when it fills up?

# File System Caching

- Replacement policy?  LRU
  - Can afford overhead full LRU implementation
  - Advantages:
    » Works very well for name translation
    » Works well in general as long as memory is big enough to accommodate a host's working set of files
  - Disadvantages:
    » Fails when some application scans through file system, thereby flushing the cache with data used only once
    » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    » File system can discard blocks as soon as they are used

# File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache $\Rightarrow$ won't be able to run many applications
  - Too little memory to file system cache $\Rightarrow$ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced

# File System Prefetching

- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request
  - Elevator algorithm can efficiently interleave prefetches from concurrent applications
- How much to prefetch?
  - Too much prefetching imposes delays on requests by other applications
  - Too little prefetching causes many seeks (and rotational delays) among concurrent file requests

# Delayed Writes

- Buffer cache is a writeback cache (writes are termed "Delayed Writes")

- `write()` copies data from user space to kernel buffer cache
  - Quick return to user space

- `read()` is fulfilled by the cache, so `read`s see the results of `write`s
  - Even if the data has not reached disk

- When does data from a `write` syscall finally reach disk?
  - When the buffer cache is full (e.g., we need to evict something)
  - When the buffer cache is flushed periodically (in case we crash)

# Delayed Writes (Advantages)

- Performance advantage: return to user quickly without writing to disk!

- Disk scheduler can efficiently order lots of requests
  - Elevator Algorithm can rearrange writes to avoid random seeks
- Delay block allocation:
  - May be able to allocate multiple blocks at same time for file, keep them contiguous
- Some files never actually make it all the way to disk
  - Many short-lived files!

# Buffer Caching vs. Demand Paging

- Replacement Policy?
  - Demand Paging: LRU is infeasible; use approximation (like NRU/Clock)
  - Buffer Cache: LRU is OK

- Eviction Policy?
  - Demand Paging: evict not-recently-used pages when memory is close to full
  - Buffer Cache: write back dirty blocks periodically, even if used recently
    » Why? To minimize data loss in case of a crash

# Dealing with Persistent State

- Buffer Cache: write back dirty blocks periodically, even if used recently
  - Why? To minimize data loss in case of a crash
  - Linux does periodic flush every 30 seconds

- Not foolproof! Can still crash with dirty blocks in the cache
  - What if the dirty block was for a directory?
    » Lose pointer to file's inode (leak space)
    » File system now in inconsistent state ☹

# Takeaway: File systems need recovery mechanisms

# File System Summary (1/2)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called "inode"
- Naming: translating from user-visible names to actual sys resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- Look at actual file access patterns
  - Many small files, but large files take up all the space!

# File System Summary (2/3)

- 4.2 BSD Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
- NTFS: variable extents not fixed blocks, tiny files data is in header
- File layout driven by freespace management
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
  - Integrate freespace, inode table, file blocks and dirs into block group
- Deep interactions between mem management, file system, sharing
  - **mmap()**: map file or anonymous segment to memory
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)