Slide 1

CS162
Operating Systems and
Systems Programming
Lecture 22

Filesystems 2: Filesystem Design (Con't),
Filesystem Case Studies

April 11th, 2024

Prof. John Kubiatowicz

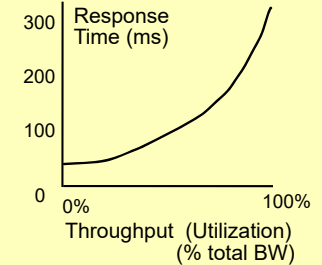http://cs162.eecs.Berkeley.edu

Slide 2

Recall: A Few Queuing Theory Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless

Arrival Rate λ → **Queue** → Service Rate $\mu = 1/T_{ser}$ → **Server**

Why does response/queueing delay grow unboundedly even though the utilization is < 1 ?

Response Time (ms): 300, 200, 100, 0 — Throughput (Utilization) (% total BW) 0% to 100%

- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results:
  - Memoryless service distribution (C = 1): (an "M/M/1 queue"):
    » $T_q = T_{ser} \times u/(1-u)$
  - General service distribution (no restrictions), 1 server (an "M/G/1 queue"):
    » $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1-u)$

4/11/2024   Kubiatowicz CS162 © UCB Spring 2024   Lec 22.2

Slide 3

Recall: I/O and Storage Layers

Application / Service

High Level I/O — *Streams*
Low Level I/O — *File Descriptors*
Syscall — open(), read(), write(), close(), … *Open File Descriptions*

What we covered in Lecture 4

File System — *Files/Directories/Indexes*

What we will cover next…

I/O Driver — *Commands and Data Transfers*

Disks, Flash, Controllers, DMA

What we just covered…

4/11/2024   Kubiatowicz CS162 © UCB Spring 2024   Lec 22.3

Slide 4

From Storage to File Systems
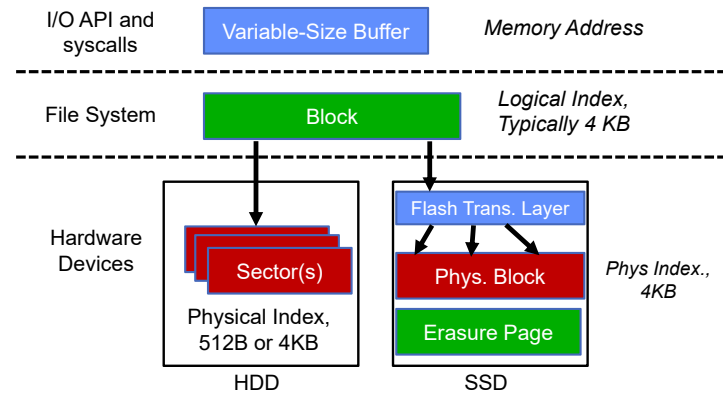
I/O API and syscalls — Variable-Size Buffer — *Memory Address*

File System — Block — *Logical Index, Typically 4 KB*

Hardware Devices:

HDD: Sector(s) — Physical Index, 512B or 4KB

SSD: Flash Trans. Layer → Phys. Block → Erasure Page — *Phys Index., 4KB*

4/11/2024   Kubiatowicz CS162 © UCB Spring 2024   Lec 22.4

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 $\geq$ sector size; in UNIX, block size is 4KB

Translation from User to System View
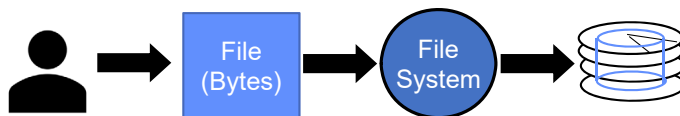
- What happens if user says: "give me bytes 2 – 12?"
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about writing bytes 2 – 12?
  - Fetch block, modify relevant portion, write out block
- Everything inside file system is in terms of whole-size blocks
  - Actual disk I/O happens in blocks
  - read/write smaller than block size needs to translate and buffer

Disk Management

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

- The disk is accessed as linear array of sectors
- How to identify a sector?
  - Physical position
    » Sectors is a vector [cylinder, surface, sector]
    » Not used anymore
    » OS/BIOS must deal with bad sectors
  - Logical Block Addressing (LBA)
    » Every sector has integer address
    » Controller translates from address $\Rightarrow$ physical position
    » Shields OS from structure of disk

What Does the File System Need?

- Track free disk blocks
  - Need to know where to put newly written data
- Track which blocks contain data for which files
  - Need to know where to read a file from
- Track files in a directory
  - Find list of file's blocks given its name
- Where do we maintain all of this?
  - Somewhere on disk

Data Structures on Disk

- Different than data structures in memory
  - Must load from disk into memory to manipulate
  - Modifications to disk data are *really* expensive, so only change when needed
- Access a block at a time
  - Can't efficiently read/write a single word
  - Have to read/write full block containing it
  - Ideally want sequential access patterns
- Durability
  - Ideally, file system is in meaningful state upon shutdown
  - This obviously isn't always the case…

FILE SYSTEM DESIGN

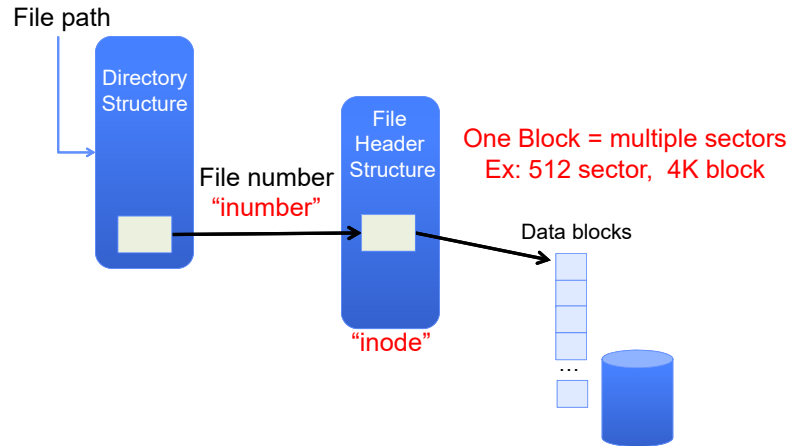## Critical Factors in File System Design

- (Hard) Disks 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
- Size is determined as they are used !!!
  - Can write (or read zeros) to expand the file
  - Start small and grow, need to make room
- Organized into directories
  - What data structure (on disk) for that?
- Need to carefully allocate / free blocks
  - Such that access remains efficient

Components of a File System

File path

Directory Structure

File Header Structure

File number
"inumber"

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

Data blocks

"inode"

...

Recall: Abstract Representation of a Process

Process

Thread's Regs

...

Address Space (Memory)

User Space

Kernel Space

File Descriptors
3

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Open File Description

File: foo.txt
Position: 100

Suppose that we execute
open("foo.txt")
and that the result is 3

Next, suppose that we execute
read(3, buf, 100)
and that the result is 100

Components of a File System

Process

Thread's Regs

...

Address Space (Memory)

User Space

Kernel Space

File Descriptors
3

Open File Description

File: foo.txt
inumber
Position: 100

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Open file description is better described as remembering the **inumber (file number)** of the file, not its name

Components of a File System

file name offset → **d**irectory structure → *file number offset* → index structure ("inode") → *storage block*

- Open performs *Name Resolution*
  - Translates path name into a "file number"
- Read and Write operate on the file number
  - Use file number as an "index" to locate the blocks

- **4 components:**
  - **directory, index structure, storage blocks, free space map**

Administrivia

- Homework 5: RPC deadline this Thursday (4/12)
- Project 3: Design doc due Monday (4/15)
- Midterm 3: April 25th
  - Everything fair game with focus on last 1/3 of class
  - Three *hand-written* cheat-sheets, double sided
- Class attendance: No credit for people who use the same photo!
- Data4All@Berkeley: Tomorrow (Friday!)
  - Friday 4/12, 12:00-1:00 in Wozniak lounge (MOVED!)
  - Undergraduate or Masters students interested in Systems broadly defined (DB, Arch, Sec, Networking, Systems, etc.) who identify as an URM in Computer Science
  - Come by for free lunch to meet fellow students
    » Sign up – look for link on Ed
  - Talk to relevant faculty, discuss possible classes, research opportunities in systems, as well as the best pizza topping!

https://tinyurl.com/bdhx8hfc

## How to get the File Number?

- Look up in *directory structure*

- A directory is a file containing <file_name : file_number> mappings
  - File number could be a file or another directory
  - Operating system stores the mapping in the directory in a format it interprets
  - Each <file_name : file_number> mapping is called a directory entry

- Process isn't allowed to read the raw bytes of a directory
  - The `read` function doesn't work on a directory
  - Instead, see `readdir`, which iterates over the map without revealing the raw bytes

- Why shouldn't the OS let processes read/write the bytes of a directory?
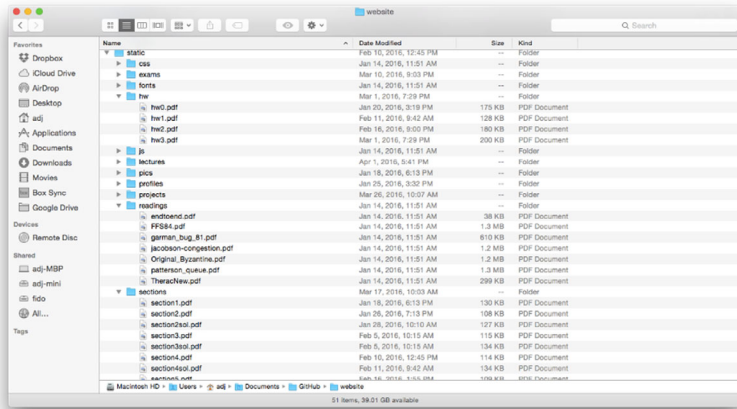
Directories

Directory Abstraction

- Directories are specialized files
  - Contents: **List of pairs <file name, file number>**
- System calls to access directories
  - `open` / `creat` / `readdir` 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
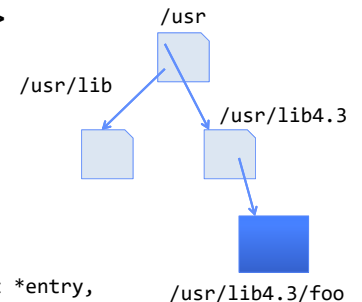
Directory Structure

- How many disk accesses to resolve "/my/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data block for root
    - » Table of file name/index pairs.
    - » Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data block for "my"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"
- Current working directory: Per-address-space pointer to a directory used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

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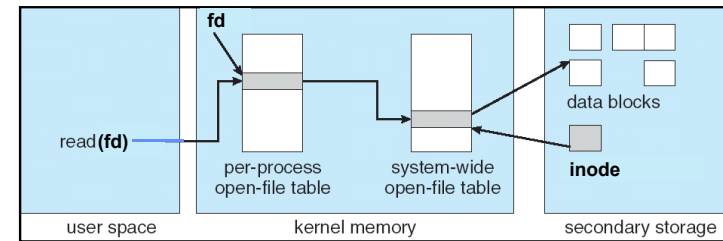
- Open syscall: find inode on disk from pathname (traversing directories)
  - Create "in-memory inode" in system-wide open file table
  - One entry in this table no matter how many instances of the file are open
- Read/write syscalls look up in-memory inode using the file handle

Characteristics of Files

A Five-Year Study of File-System Metadata

Published in FAST 2007

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
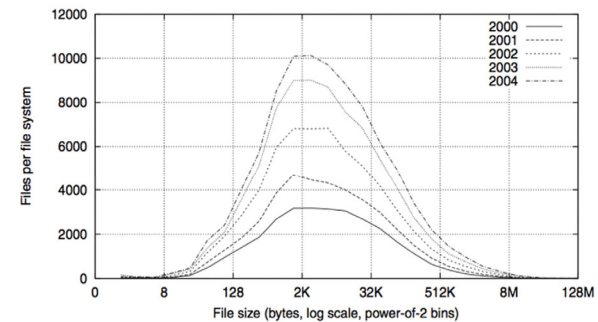Microsoft Research

Observation #1: Most Files Are Small

Fig. 2. Histograms of files by size.

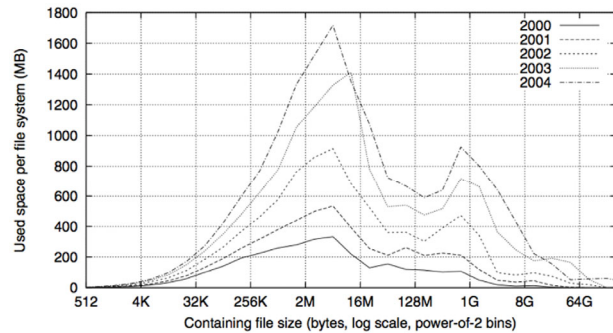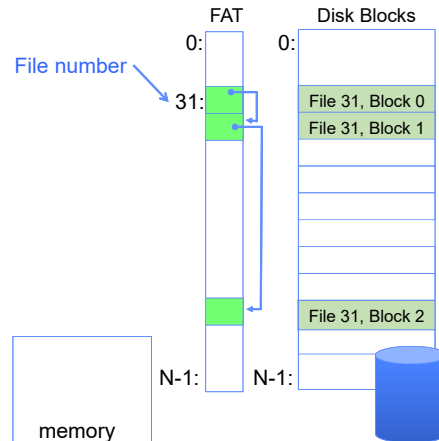## Observation #2: Most Bytes are in Large Files

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

CASE STUDY:
FAT: FILE ALLOCATION TABLE

- MS-DOS, 1977
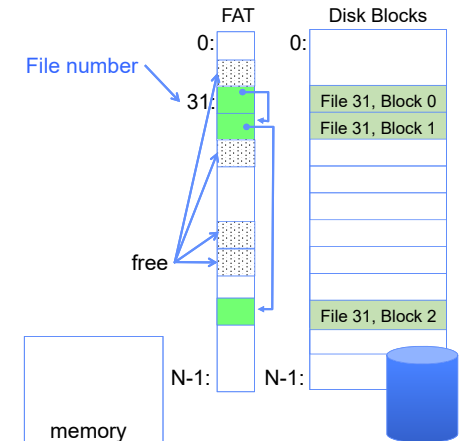- Still widely used!

FAT (File Allocation Table)

- Assume (for now) we have a way to translate a path to a "file number"
  - i.e., a directory structure
- Disk Storage is a collection of Blocks
  - Just hold file data (offset o = < B, x >)
- Example: file_read 31, < 2, x >
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into memory

FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list

FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  – Could require scan to find
  – Or, could use a free list

FAT   Disk Blocks

File number

0:   0:
31:   File 31, Block 0
  File 31, Block 1
free
  File 31, Block 2
N-1:   N-1:
memory

## FAT (File Allocation Table)

- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  – Could require scan to find
  – Or, could use a free list
- Ex: file_write(31, < 3, y >)
  – Grab free block
  – Linking them into file

FAT   Disk Blocks

File number

0:   0:
31:   File 31, Block 0
  File 31, Block 1
free
  File 31, Block 3
  File 31, Block 2
N-1:   N-1:
memory

FAT (File Allocation Table)

- Where is FAT stored?
  – On disk
- How to format a disk?
  – Zero the blocks, mark FAT entries "free"
- How to quick format a disk?
  – Mark FAT entries "free"

- Simple: can implement in device firmware

FAT   Disk Blocks

File #1

0:   0:
31:   File 31, Block 0
  File 31, Block 1
free
File #2
  File 31, Block 3
  File 31, Block 2
N-1:   N-1:
memory

FAT: Directories

| | | file 5268830 "/home/tom" | | | | | end of file |

| Name File Number Next | . 5268830 | .. 88026158 | Music 35002320 | Work 85200219 | Free Space | foo.txt 66212871 | Free Space | |

- A directory is a file containing <file_name: file_number> mappings
- Free space for new/deleted entries
- In FAT: file attributes are kept in directory (!!!)
  – Not directly associated with the file itself
- Each directory a linked list of entries
  – Requires linear search of directory to find particular entry
- Where do you find root directory ("/")?
  – At well-defined place on disk
  – For FAT, this is at block 2 (there are no blocks 0 or 1)
  – Remaining directories

FAT Discussion

Suppose you start with the file number:
- Time to find block?
- Block layout for file?
- Sequential access?
- Random access?
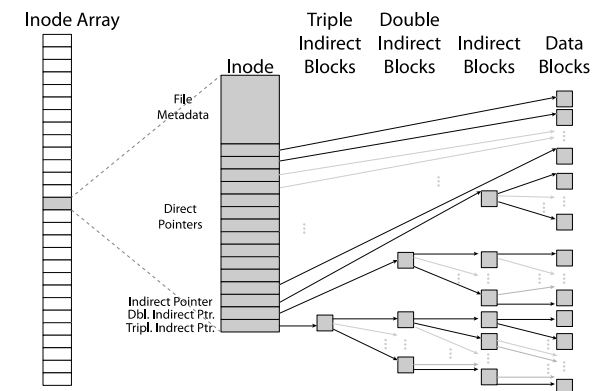- Fragmentation?
- Small files?
- Big files?
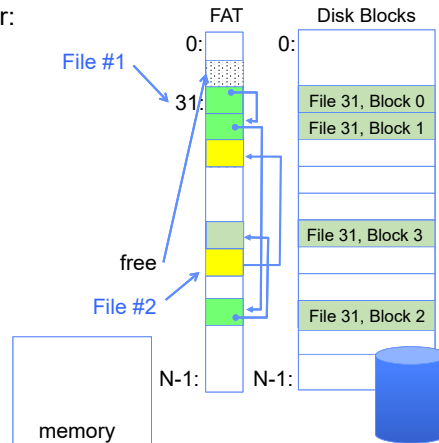
CASE STUDY:
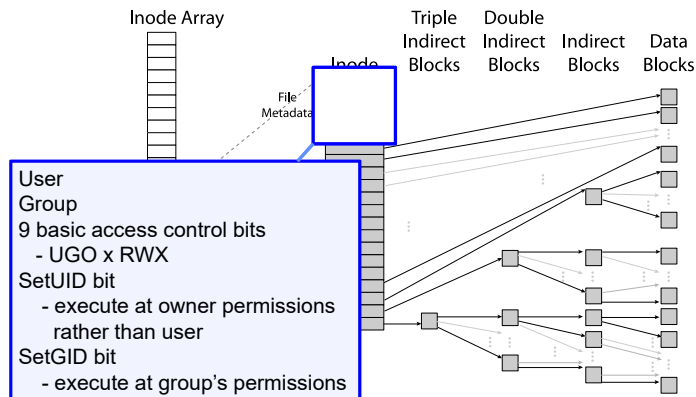# UNIX FILE SYSTEM (BERKELEY FFS)

Inodes in Unix (Including Berkeley FFS)

- File Number is index into set of inode arrays
- Index structure is an array of *inodes*
  - File Number (inumber) is an index into the array of inodes
  - Each inode corresponds to a file and contains its metadata
    » So, things like read/write permissions are stored with *file,* not in directory
    » Allows multiple names (directory entries) for a file
- Inode maintains a multi-level tree structure to find storage blocks for files
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks

- Original *inode* format appeared in BSD 4.1 (more following)
  - Berkeley Standard Distribution Unix!
  - Part of your heritage!
  - Similar structure for Linux Ext 2/3

Inode Structure

File Atributes

Inode Array

Inode

File Metadata

Triple Indirect Blocks Double Indirect Blocks Indirect Blocks Data Blocks

User
Group
9 basic access control bits
  - UGO x RWX
SetUID bit
  - execute at owner permissions
    rather than user
SetGID bit
  - execute at group's permissions

Small Files: 12 Pointers Direct to Data Blocks

Direct pointers

4kB blocks $\Rightarrow$ sufficient for files up to 48KB

Inode

Triple Indirect Blocks Double Indirect Blocks Indirect Blocks Data Blocks

File Metadata

Direct Pointers

Indirect Pointer
Dbl. Indirect Ptr.
Tripl. Indirect Ptr.

Fig. 2. Histograms of files by size.
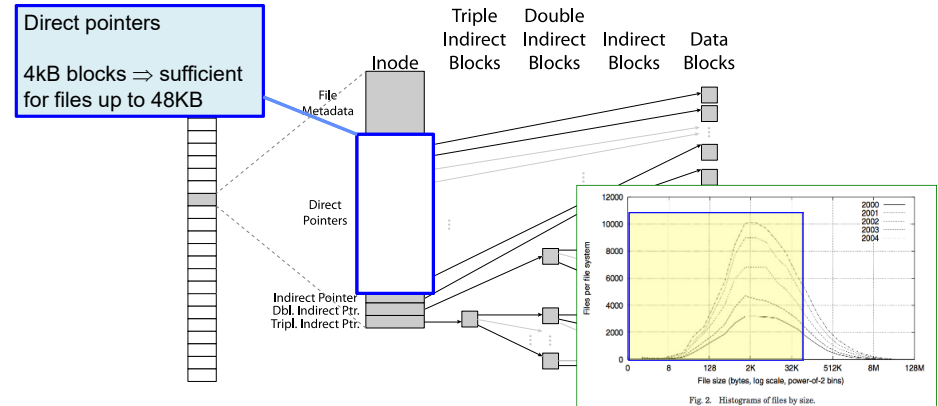
Large Files: 1-, 2-, 3-level indirect pointers

Indirect pointers
  - point to a disk block
    containing only pointers
  - 4 kB blocks => 1024 ptrs
    => 4 MB @ level 2
    => 4 GB @ level 3
    => 4 TB @ level 4

Inode

Triple Indirect Blocks Double Indirect Blocks Indirect Blocks Data Blocks

Direct Pointers

Indirect Pointer
Dbl. Indirect Ptr.
Tripl. Indirect Ptr.

48 KB

+4 MB

+4 GB

+4 TB

Fig. 4. Histograms of bytes by containing file size.
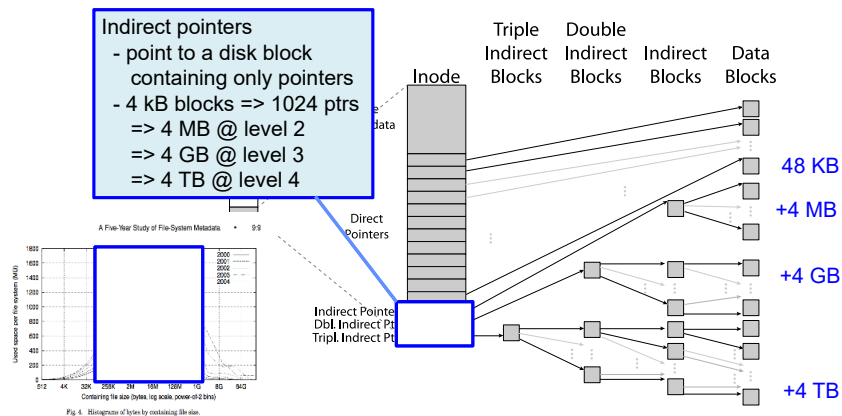
Putting it All Together: On-Disk Index

- 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

Inode Array

Inode

File Metadata

Triple Indirect Blocks Double Indirect Blocks Indirect Blocks Data Blocks

Direct Pointers

Indirect Pointer
Dbl. Indirect Ptr.
Tripl. Indirect Ptr.
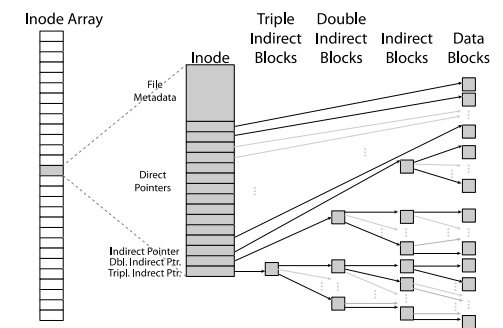
Recall: Critical Factors in File System Design

- **(Hard) 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
- Size is determined as they are used !!!
  - Can write (or read zeros) to expand the file
  - Start small and grow, need to make room
- Organized into directories
  - What data structure (on disk) for that?
- Need to carefully allocate / free blocks
  - Such that access remains efficient

Recall: Magnetic Disks

- Cylinders: all the tracks under the head at a given point on all surfaces
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector) under r/w head

Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time

Request → Software Queue (Device Driver) → Hardware Controller → Media Time (Seek+Rot+Xfer) → Result
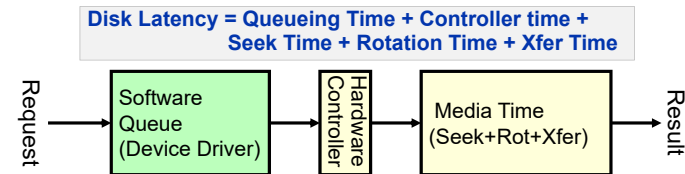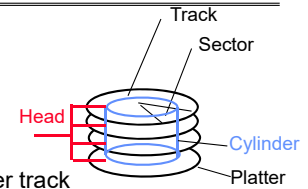
Fast File System (BSD 4.2, 1984)

- 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)

FFS Changes in Inode Placement: Motivation

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Fixed size, set when disk is formatted
    - » At formatting time, a fixed number of inodes are created
    - » Each is given a unique number, called an "inumber"

- Problem #1: Inodes all in one place (outer tracks)
  - Head crash potentially destroys all files by destroying inodes
  - Inodes not close to the data that the point to
    - » To read a small file, seek to get header, seek back to data

- Problem #2: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
 - Makes it hard to optimize for performance

FFS Locality: Block Groups

- The UNIX BSD 4.2 (FFS) distributed the header information (inodes) closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file
  - makes an "ls" of that directory run very fast
- File system volume divided into set of block groups
  - Close set of tracks
- Data blocks, metadata, and free space interleaved within block group
  - Avoid huge seeks between user data and system structure
- Put directory and its files in common block group
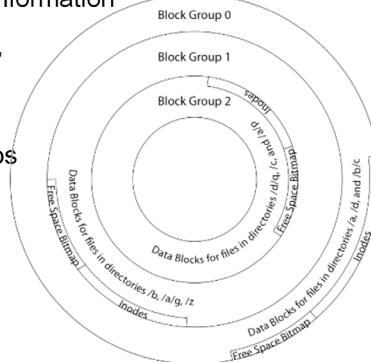
FFS Locality: Block Groups (Con't)

- First-Free allocation of new file blocks
  - To expand file, first try successive blocks in bitmap, then choose new range of blocks
  - Few little holes at start, big sequential runs at end of group
  - Avoids fragmentation
  - Sequential layout for big files
- Important: keep 10% or more free!
  - Reserve space in the Block Group
- Summary: FFS Inode Layout Pros
  - For small directories, can fit all data, file headers, etc. in same cylinder ⇒ no seeks!
  - File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
  - Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
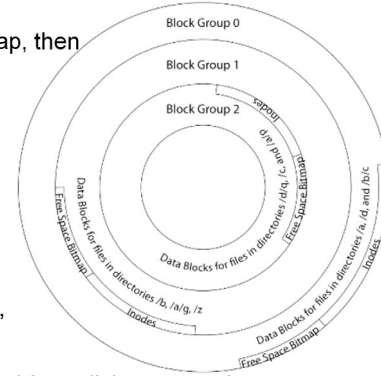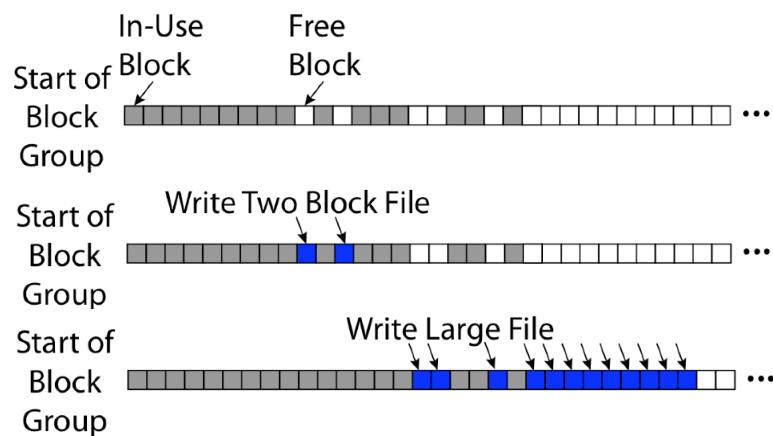
UNIX 4.2 BSD FFS First Fit Block Allocation
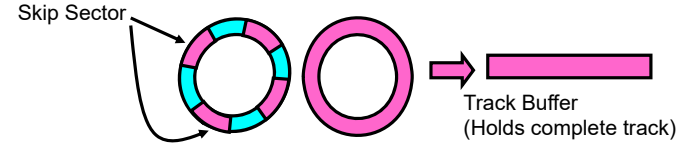
Attack of the Rotational Delay

- Problem 3: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!

  - Solution1: Skip sector positioning ("interleaving")
    » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
    » Can be done by OS or in modern drives by the disk controller
  - Solution 2: Read ahead: read next block right after first, even if application hasn't asked for it yet
    » This can be done either by OS (read ahead)
    » By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track
- Modern disks + controllers do many things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

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

---

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
- Ext3: Ext2 with Journaling
  - Several degrees of protection with comparable overhead
  - We will talk about Journalling later
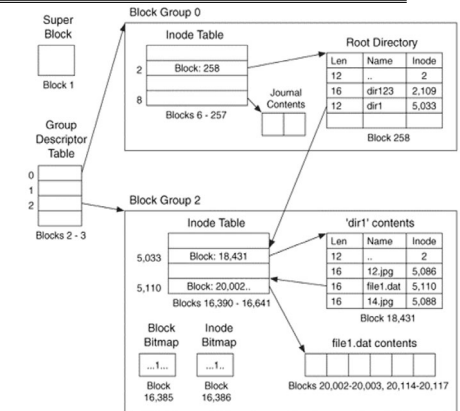
- Example: create a `file1.dat` under `/dir1/` in Ext3

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
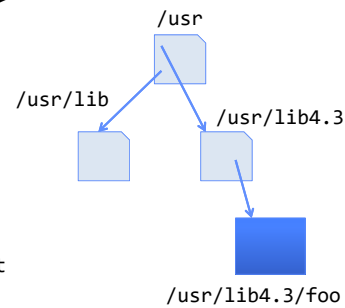
Hard Links

- 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
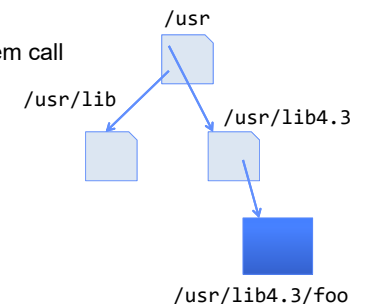
Soft Links (Symbolic Links)

- 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, dest. 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

- 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
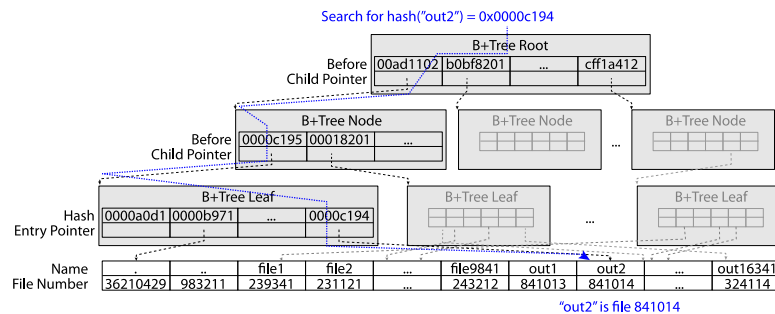- **Check permissions on the final inode and each directory's inode…**
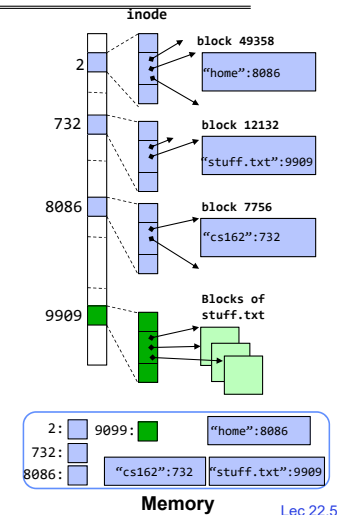
Large Directories: B-Trees (dirhash)

in FreeBSD, NetBSD, OpenBSD

Conclusion

- File System:
  – Transforms blocks into Files and Directories
  – Optimize for access and usage patterns
  – Maximize sequential access, allow efficient random access
- File (and directory) 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
- File Allocation Table (FAT) Scheme
  – Linked-list approach
  – Very widely used: Cameras, USB drives, SD cards
  – Simple to implement, but poor performance and no security
- Look at actual file access patterns
  – Many small files, but large files take up all the space!
- 4.2 BSD Fast File System: Multi-level inode header to describe files
  – Inode contains ptrs to actual blocks, indirect blocks, double indirect blocks, etc.
  – Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization