CS162
Operating Systems and Systems Programming
Lecture 18

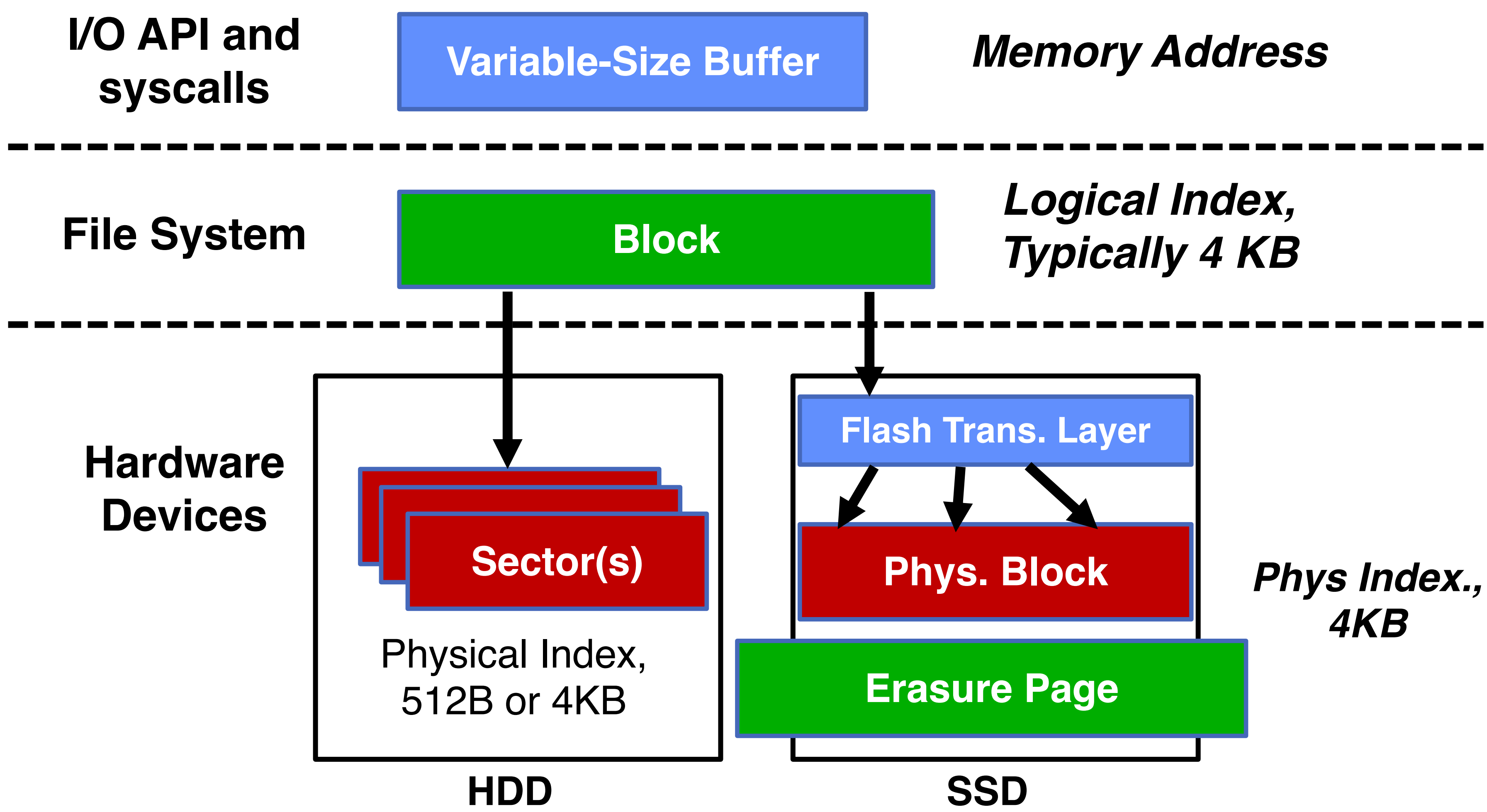
File Systems

October 31, 2019

Prof. David E. Culler
http://cs162.eecs.Berkeley.edu

Read: A&D Ch 13

Recall: From Storage to File Systems

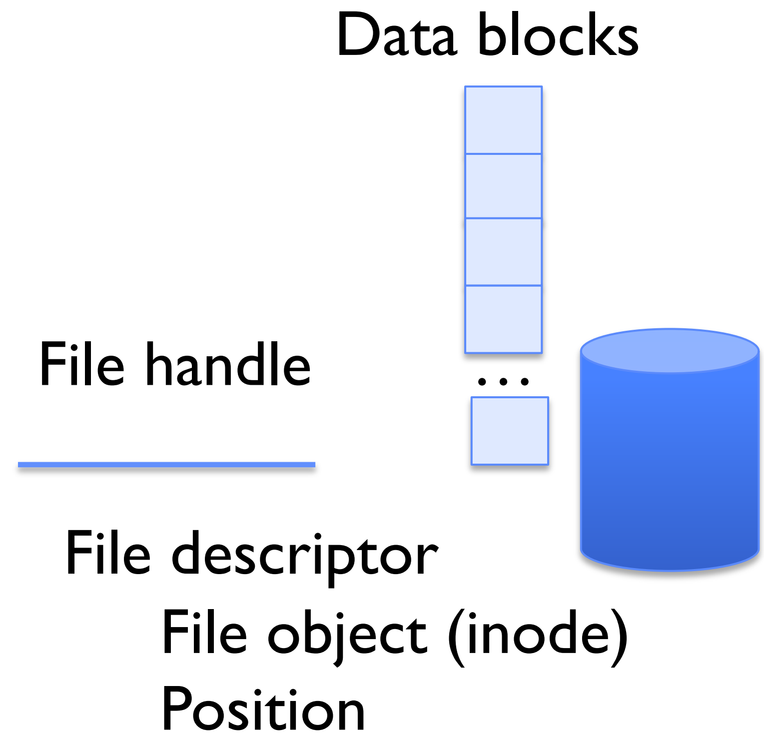


Recall: Designing a File System …

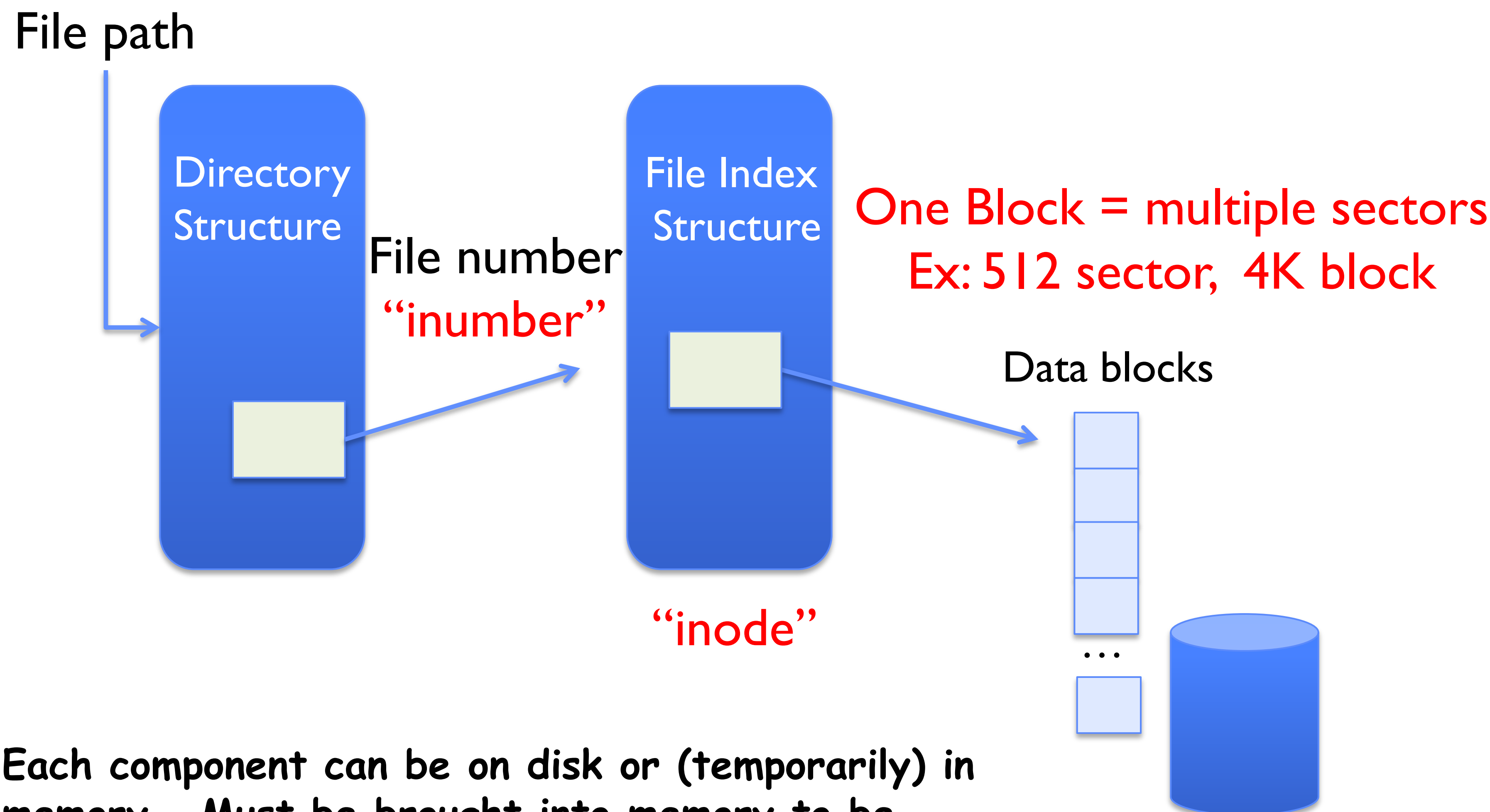
- What factors are critical to the design choices?
- Durable data store => it's all on disk
- (Hard) Disks Performance !!!
  – Maximize sequential access, minimize seeks
- Open before Read/Write
  – Can perform protection checks and look up where the actual file resource are, in advance
- Size is determined 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 allocate / free blocks
  – Such that access remains efficient

Recall: 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



Recall: Components of a File System



Each component can be on disk or (temporarily) in memory. Must be brought into memory to be accessed.

Recall: Components of a file system

file name → *file number* → *Storage block*

offset directory *offset* index structure

- 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 integer) to user process

- Read, Write, Seek, and Sync operate on handle
  - Mapped to file descriptor and to blocks

Recall: What does the file system need?

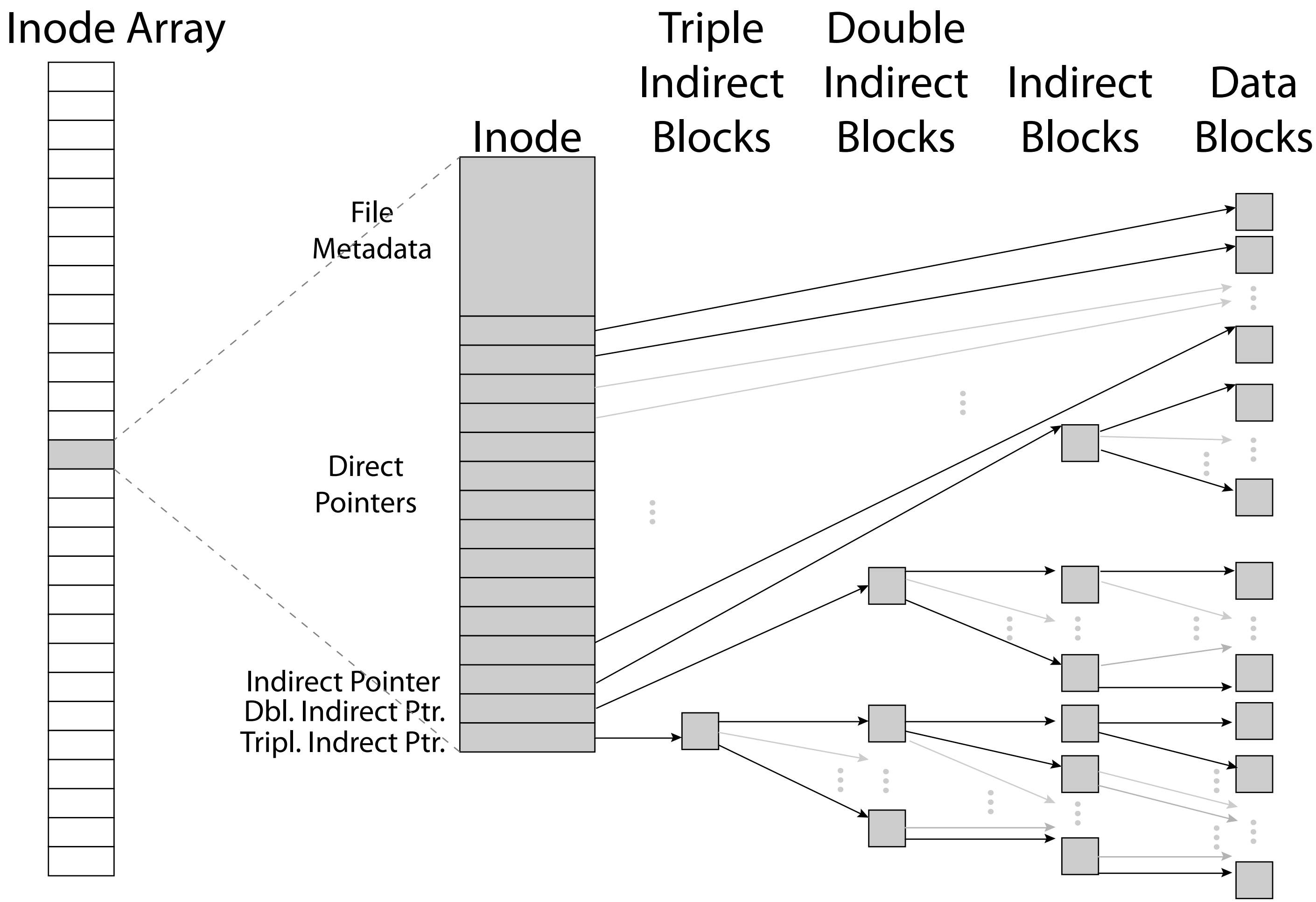
- Track free disk blocks
  - Need to know where to put newly written data
- Track which blocks contain data for which files
  - Need to know where to read a file from
- Track files in a directory
  - Find list of file's blocks given its name
- Where do we maintain all of this?
  - Somewhere on disk

Unix File System

- Original *inode* format appeared in BSD 4.1 (more following)
  - Berkeley Standard Distribution Unix (Part of your heritage)
  - Similar structure for Linux Ext2/3
- "File Number" is index into *inode array*
- Metadata associated with the file
  - Rather than in the directory that points to it
- Multi-level index structure
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks
- A ***volume*** is a collection of physical storage resources that form a logical storage device
  - Each instance of a file system manages files and directories for a volume (try **`>>> df`**)
  - A volume is ***mount***ed in existing directory structure (eg **`/home`**)

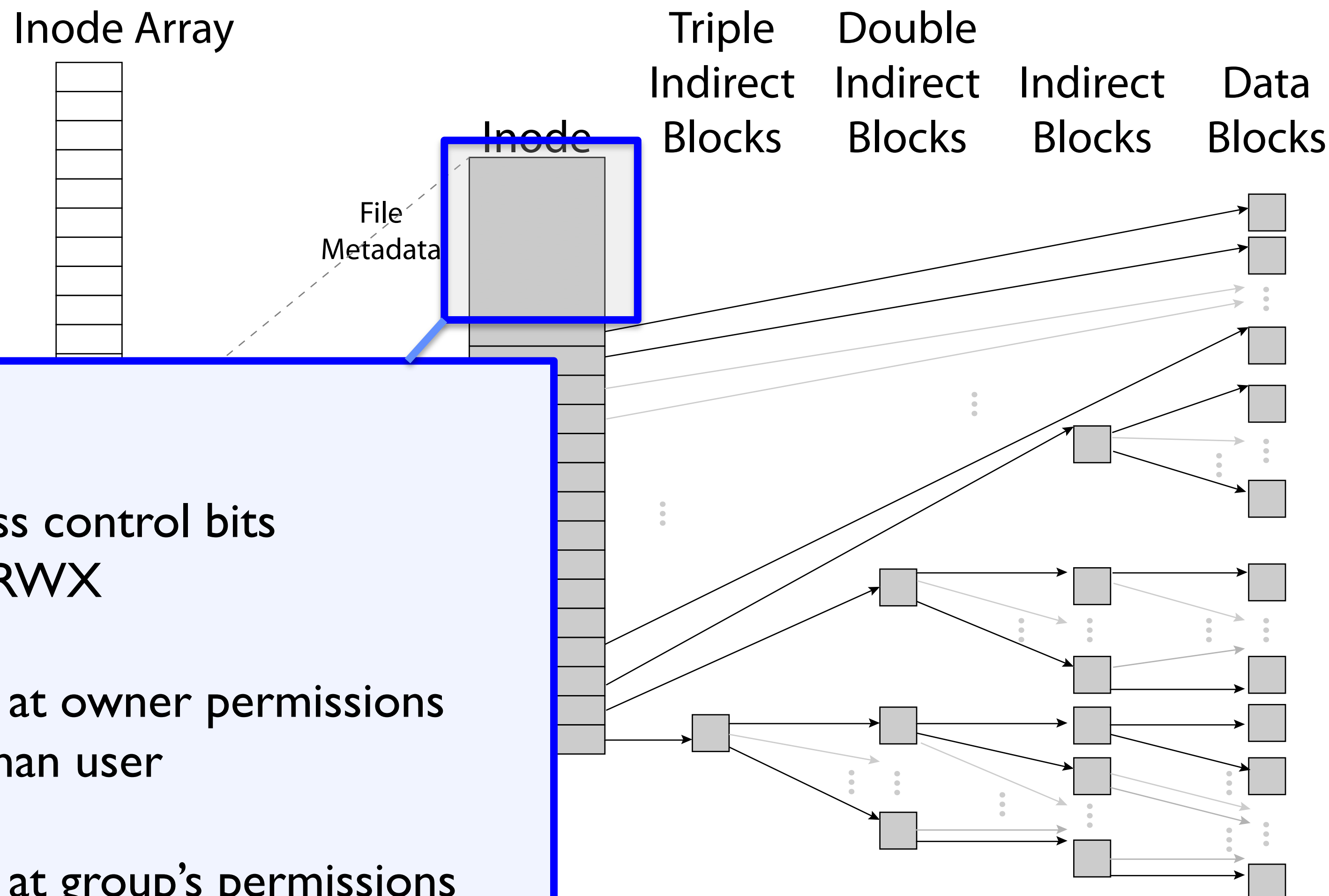
Inode Structure

- inode metadata



File Attributes

- inode metadata



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

An “almost real” file system

- Pintos: src/filesys/file.c, inode.c

```
/* An open file. */
struct file
  {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
  };
```

```
/* In-memory inode. */
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;             /* Inode content. */
  };
```

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

Characteristics of Files

A Five-Year Study of File-System Metadata

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

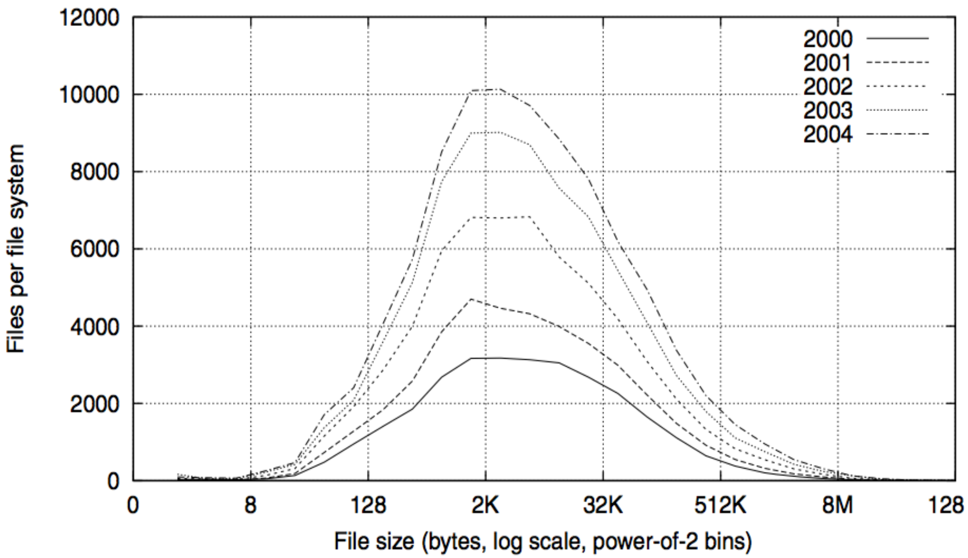


Fig. 2. Histograms of files by size.

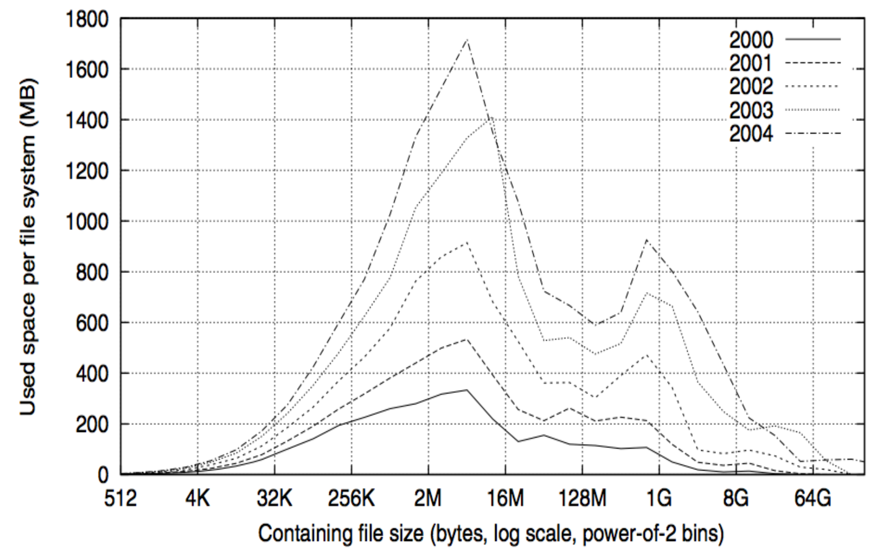


Fig. 4. Histograms of bytes by containing file size.

Characteristics of Files

- Most files are small, growing numbers of files over time

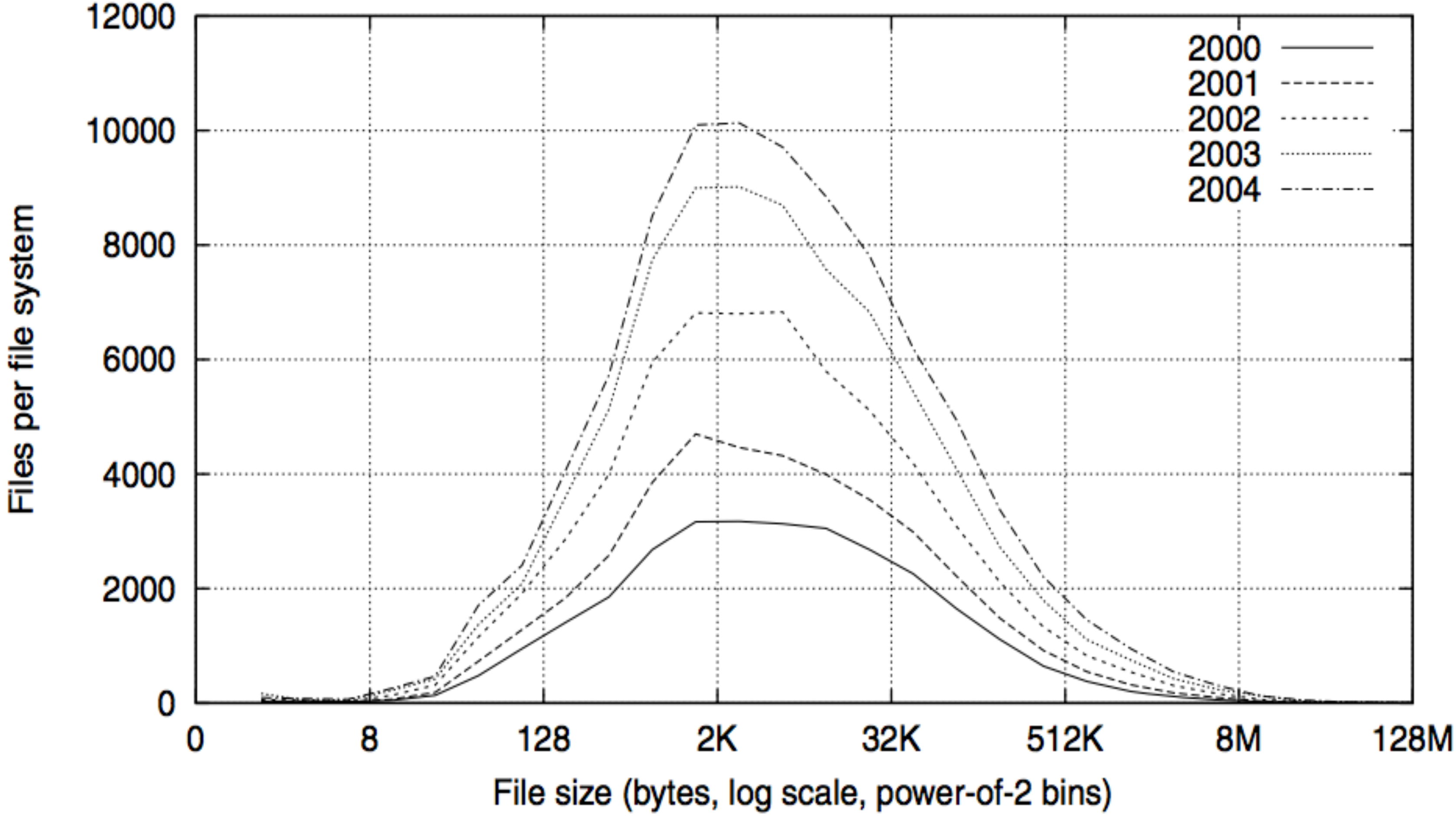


Fig. 2. Histograms of files by size.

Characteristics of Files

- Most of the space is occupied by the rare big ones

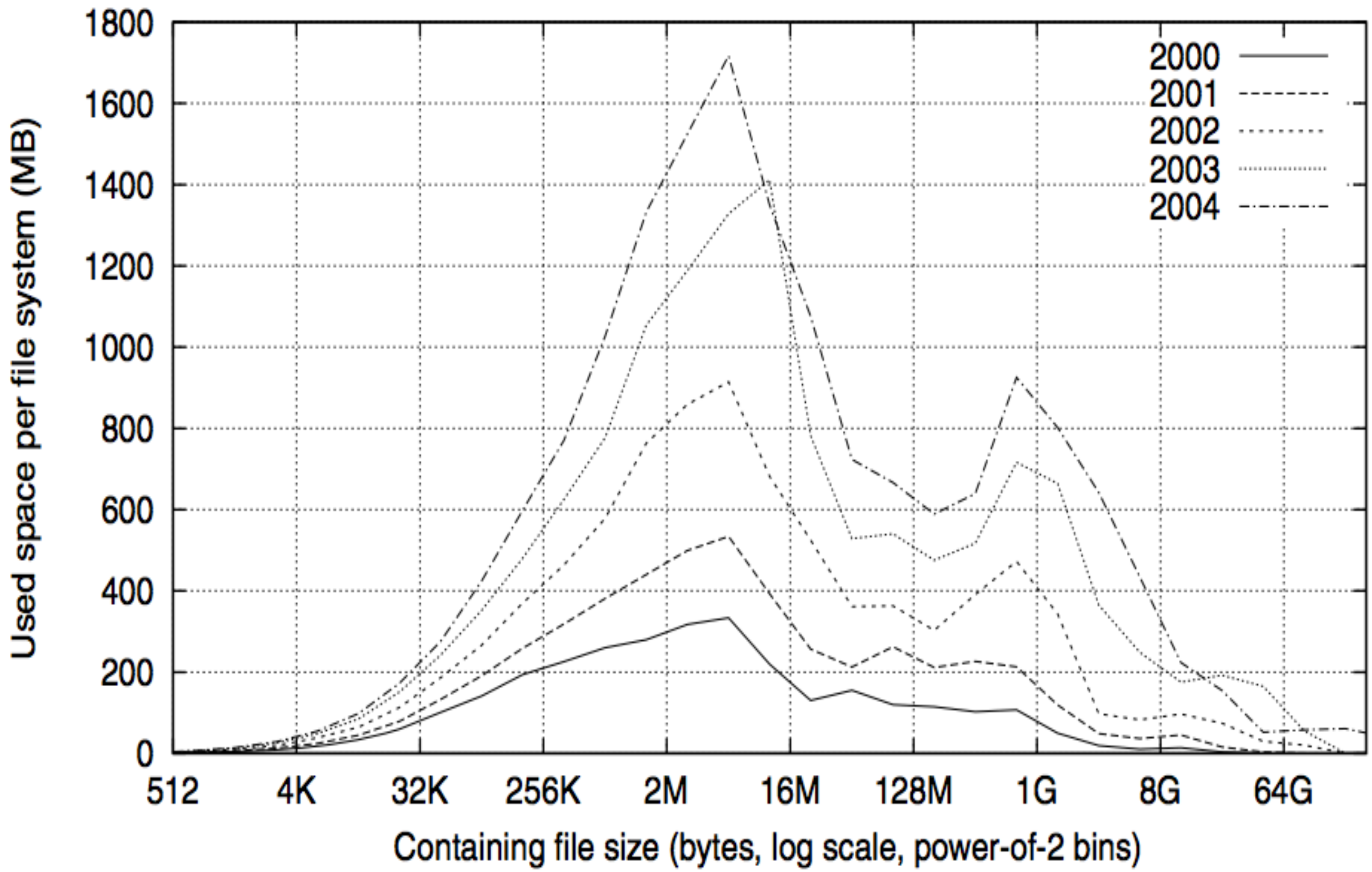


Fig. 4. Histograms of bytes by containing file size.

Data Storage

- Small files: 12 pointers direct to data blocks

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

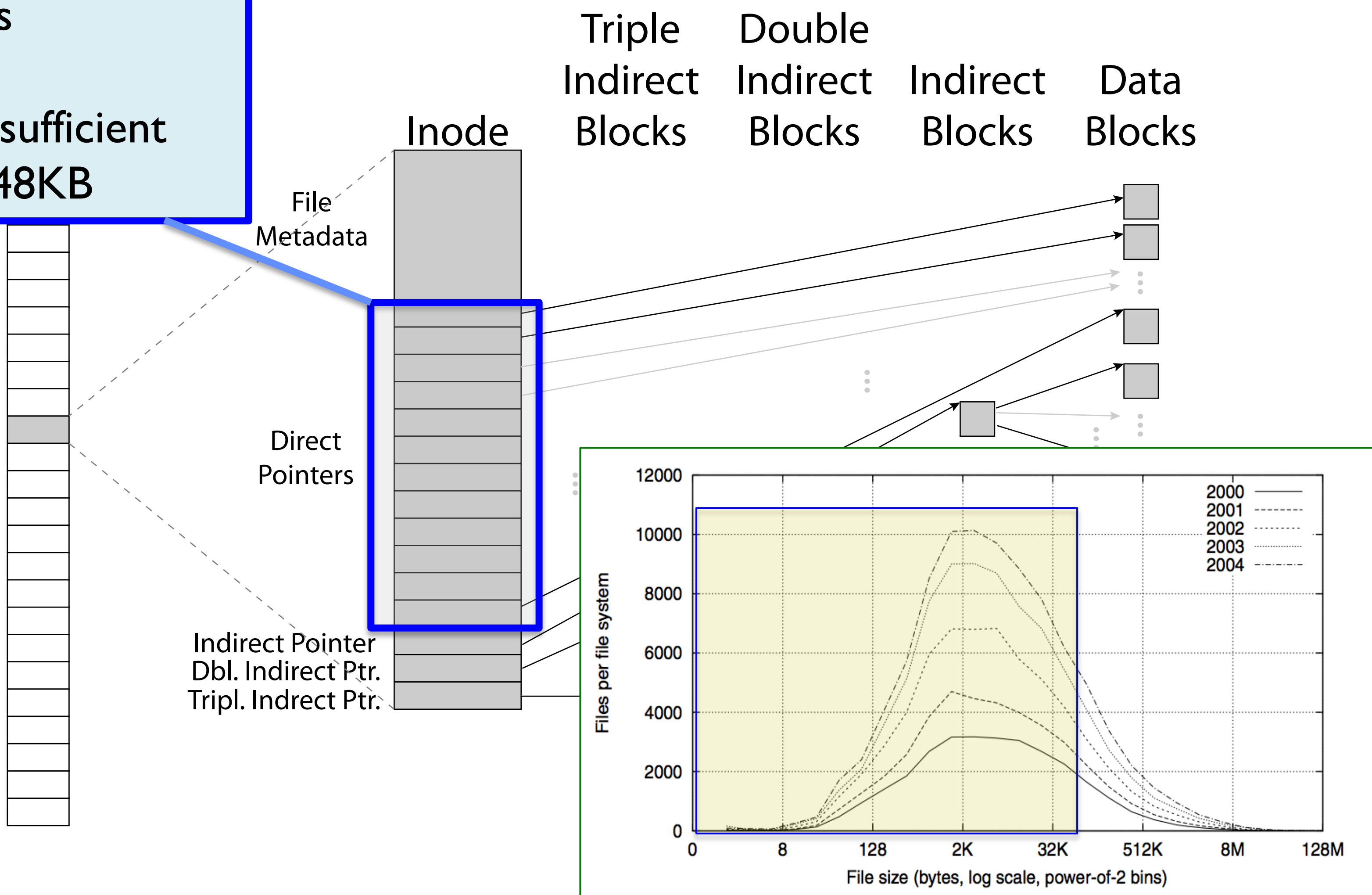
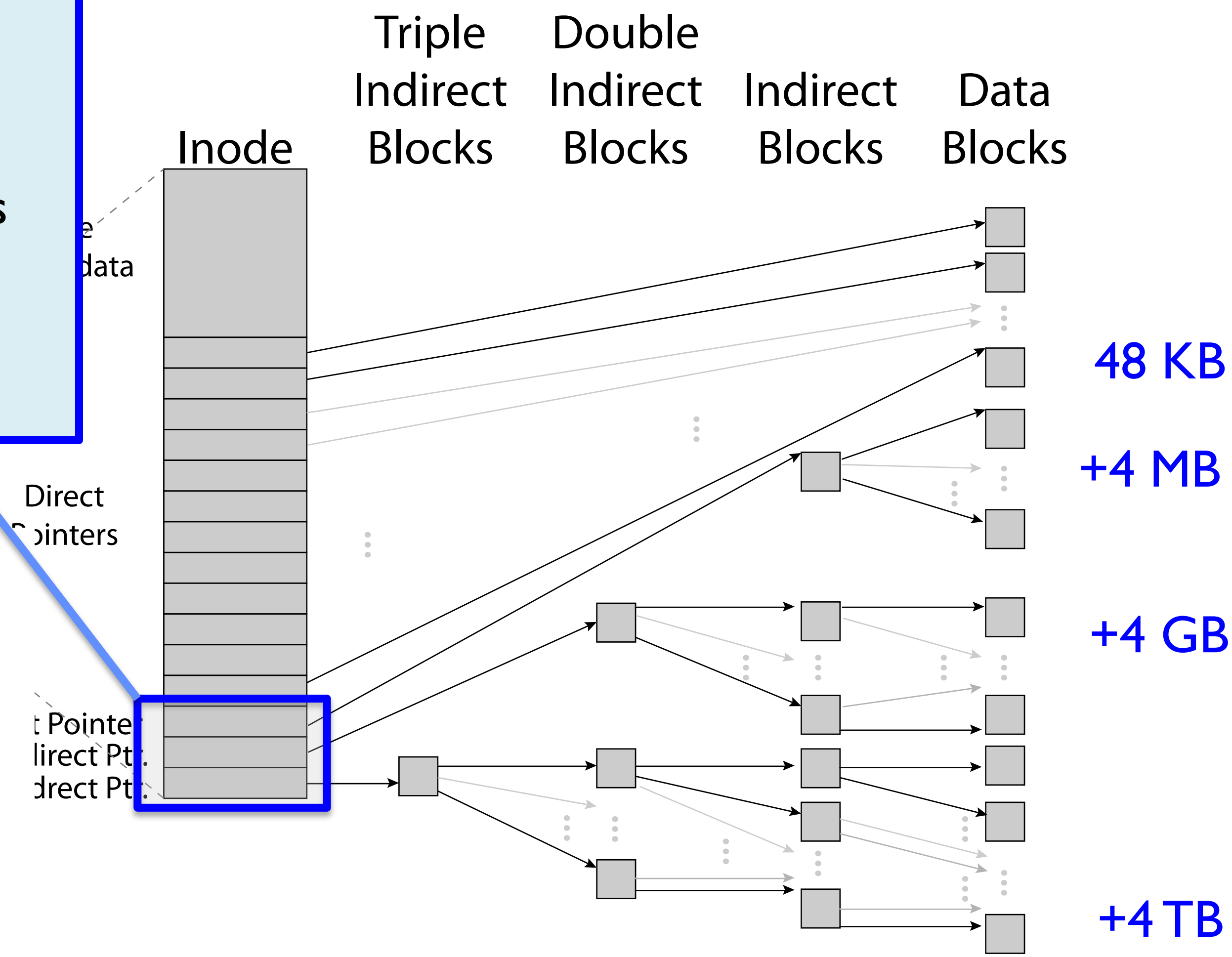


Fig. 2. Histograms of files by size.

Data Storage

- 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



A Five-Year Study of File-System Metadata • 9:9

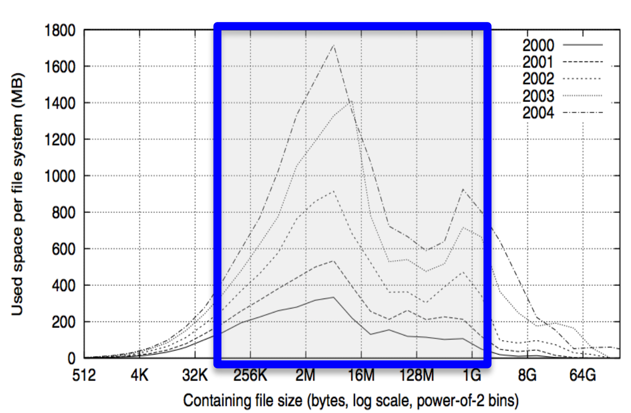
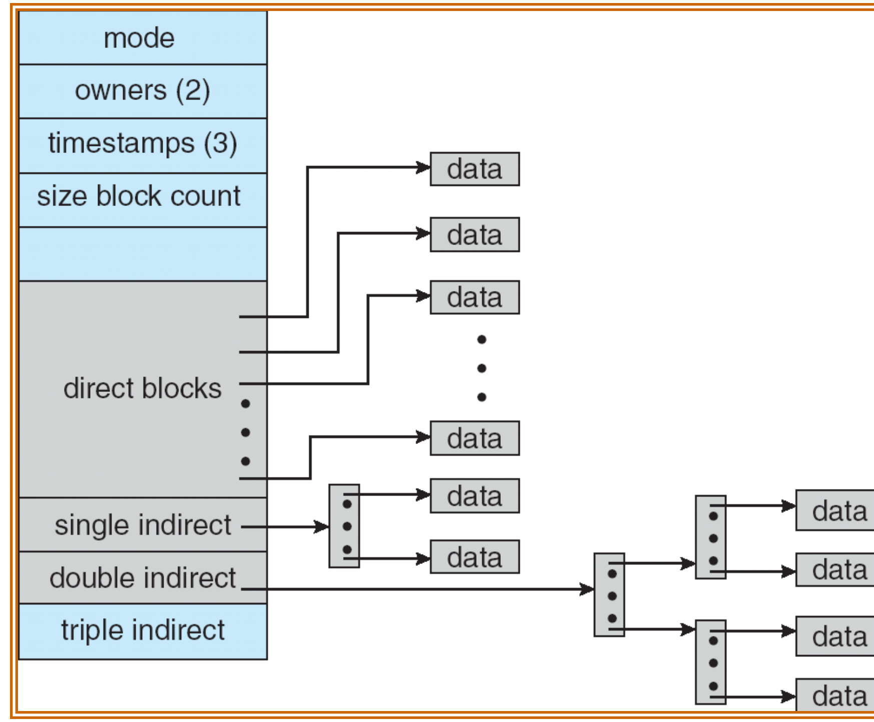


Fig. 4. Histograms of bytes by containing file size.

Multilevel Indexed Files (Original 4.1 BSD) Example

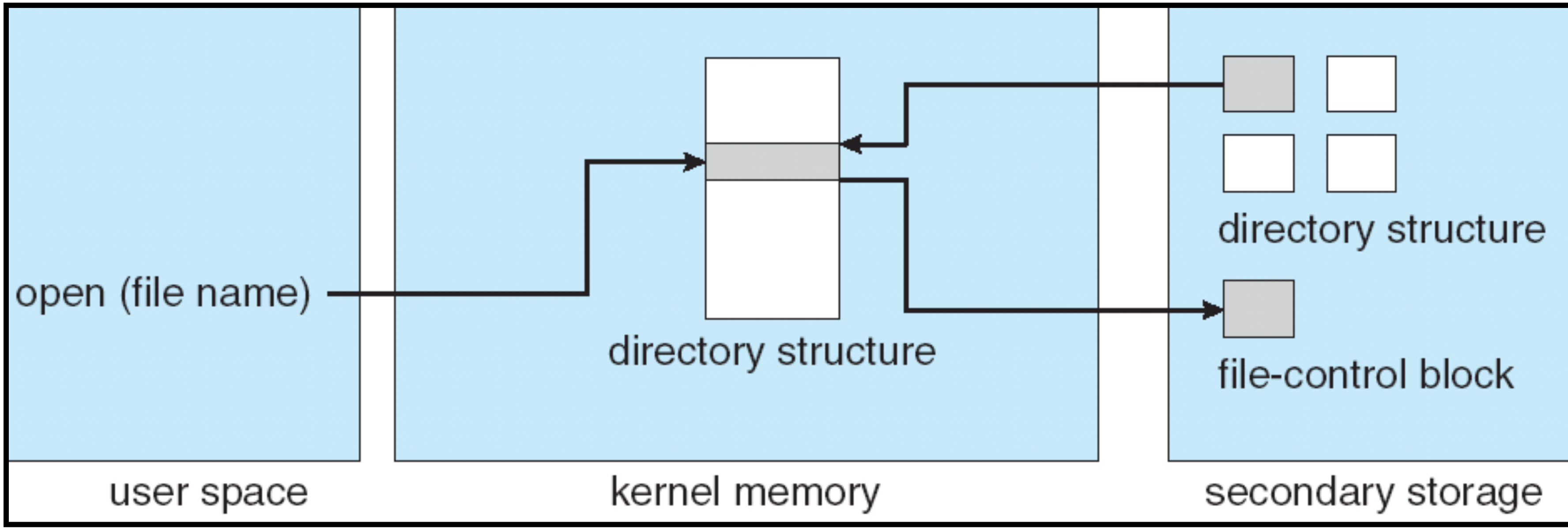


- Sample file in multilevel indexed format:
  - 10 direct ptrs, 1K blocks
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, 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 (lead to 4.2 Fast File System Optimizations)

Where are inodes Stored?

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
- Header not stored anywhere near the data blocks
  – To read a small file, seek to get header, seek back to data
- 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"
- Later versions of UNIX moved the header information to be 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 fast)

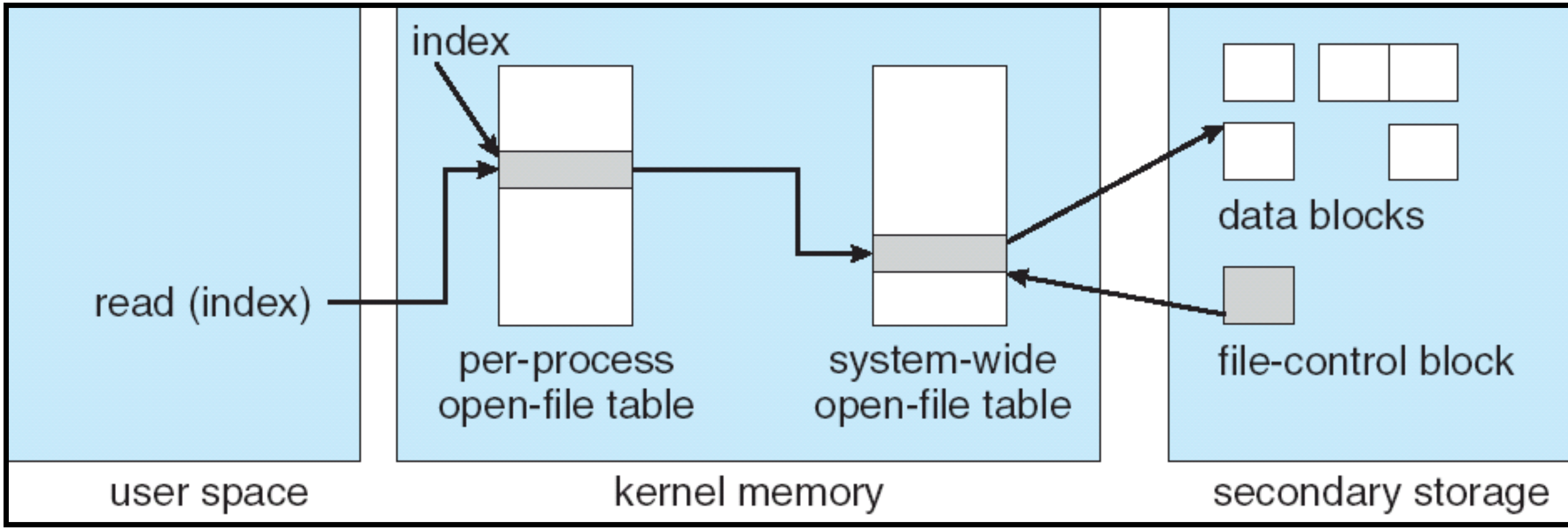
In-Memory File System Structures



- 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

*** FCB terminology from Silberschatz textbook**

In-Memory File System Structures



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

UNIX BSD 4.2 (1984) (1/2)

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from Cray Operating System:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned later)

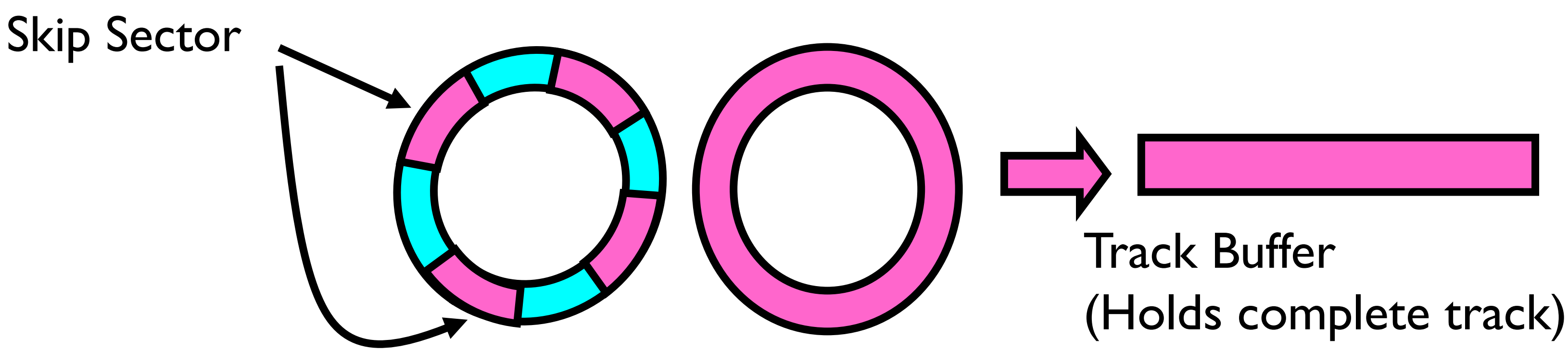
UNIX BSD 4.2 (1984) (2/2)

- Problem: 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?
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other

- Fast File System (FFS)
  - Allocation and placement policies for BSD 4.2

Attack of the Rotational Delay

- Problem 2: 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!



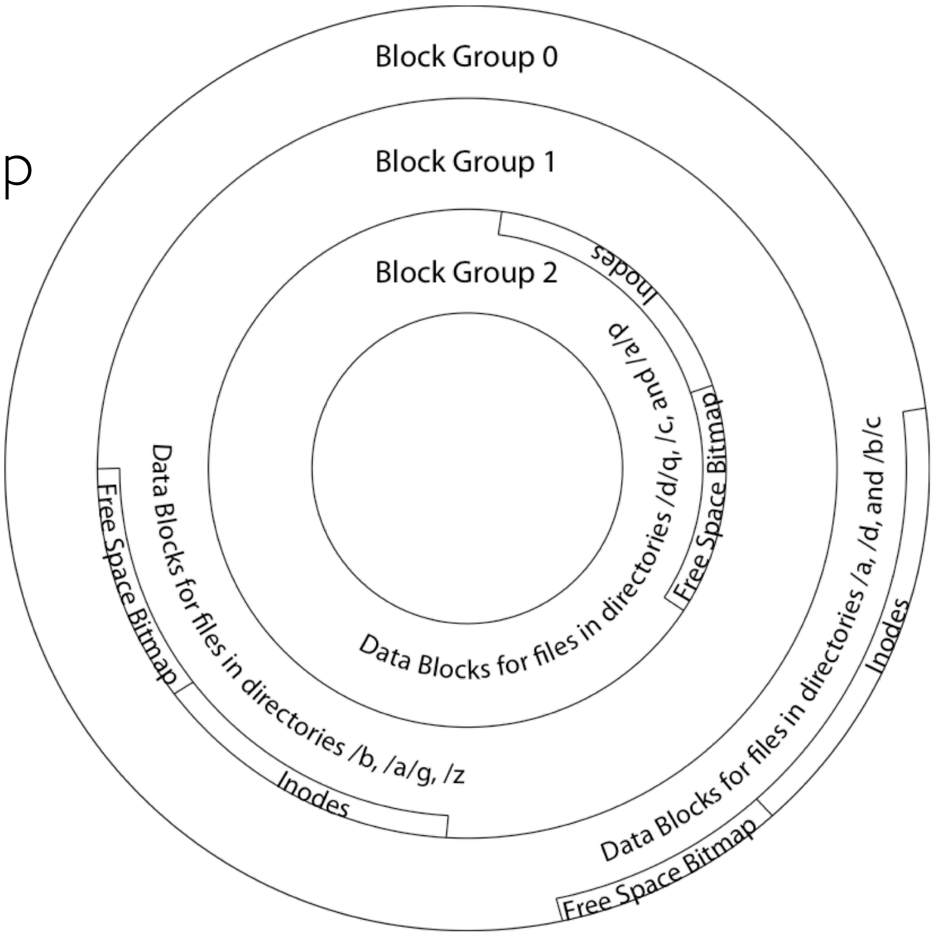
  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
    - » Can be done by OS or in modern drives by the disk controller
  - Solution 2: Read ahead: read next block right after first, 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

Where are inodes Stored?

- Later versions of UNIX moved the header information to be 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 fast)
- Pros:
  - UNIX BSD 4.2 puts bits of file header array on many cylinders
  - For small directories, can fit all data, file headers, etc. in same cylinder $\Rightarrow$ 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)
- Part of the Fast File System (FFS)
  - General optimization to avoid seeks

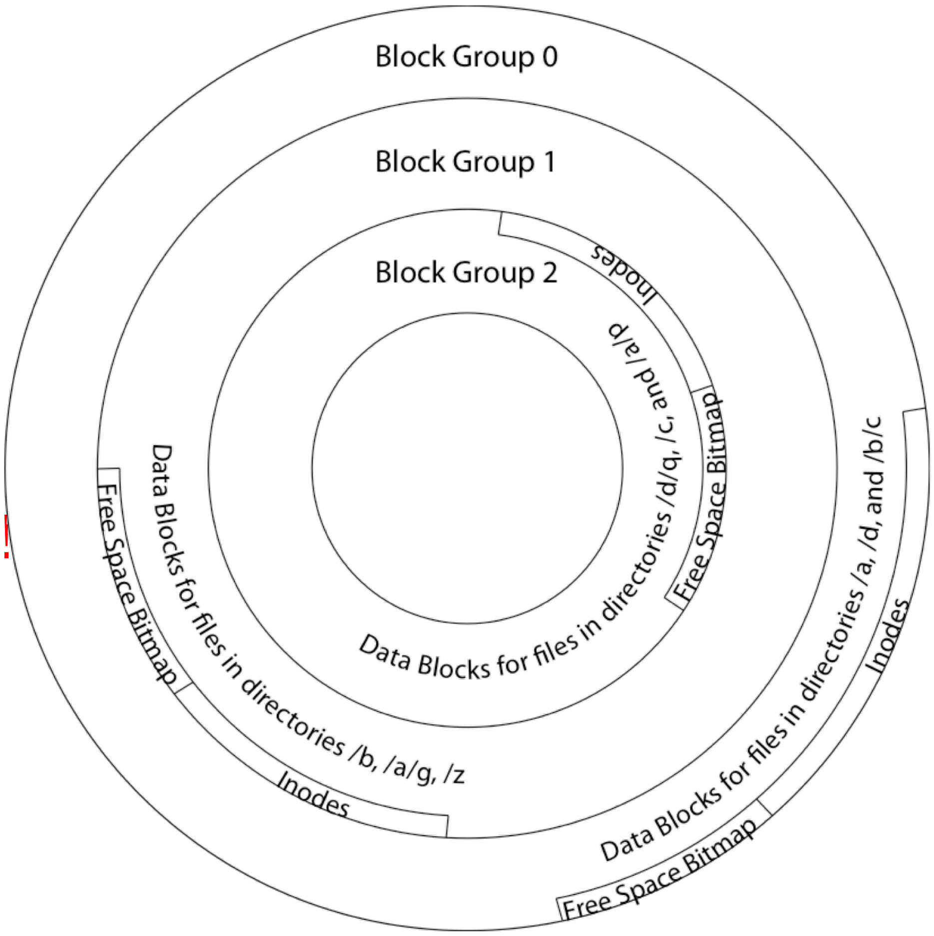
4.2 BSD Locality: Block Groups

- File system volume is divided into a 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



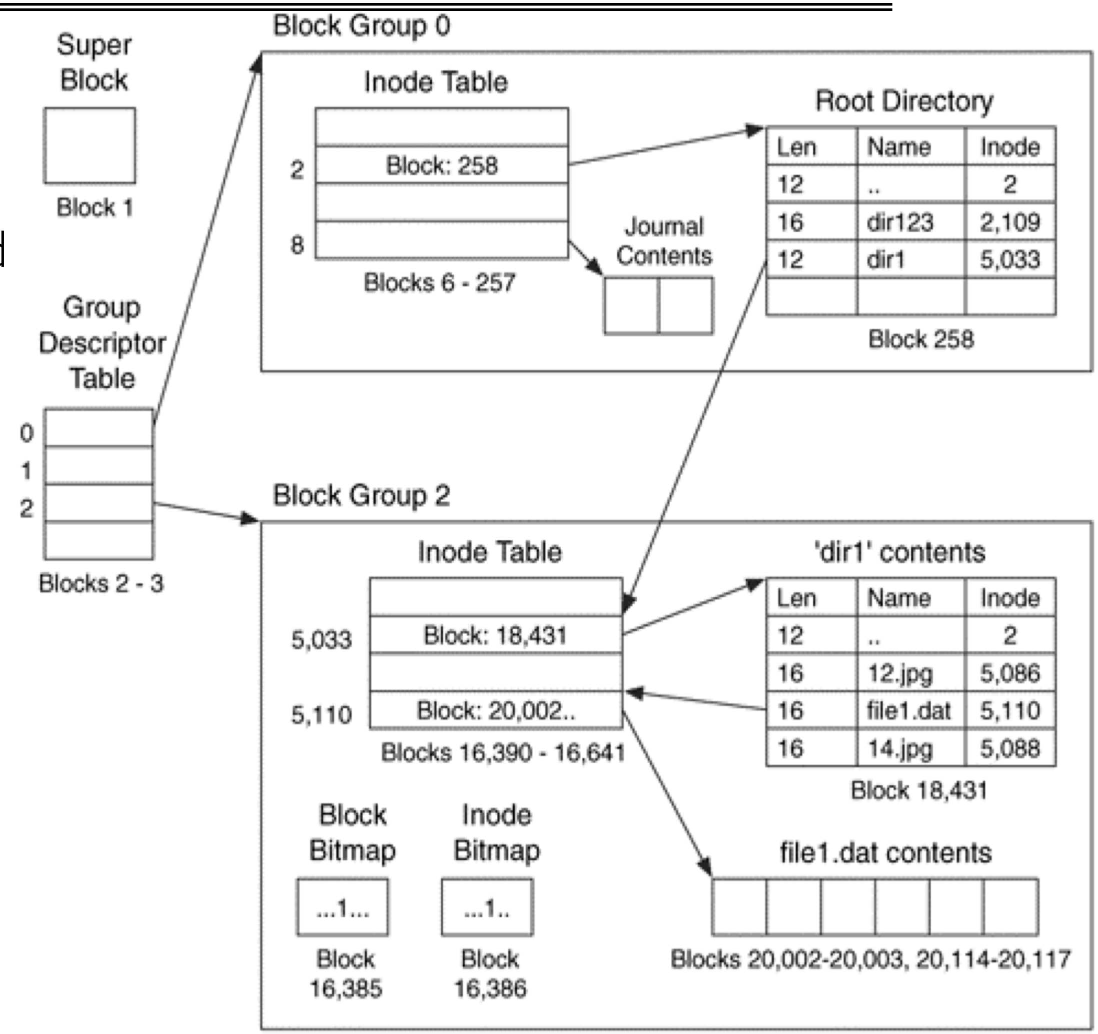
4.2 BSD Locality: Block Groups

- 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



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



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

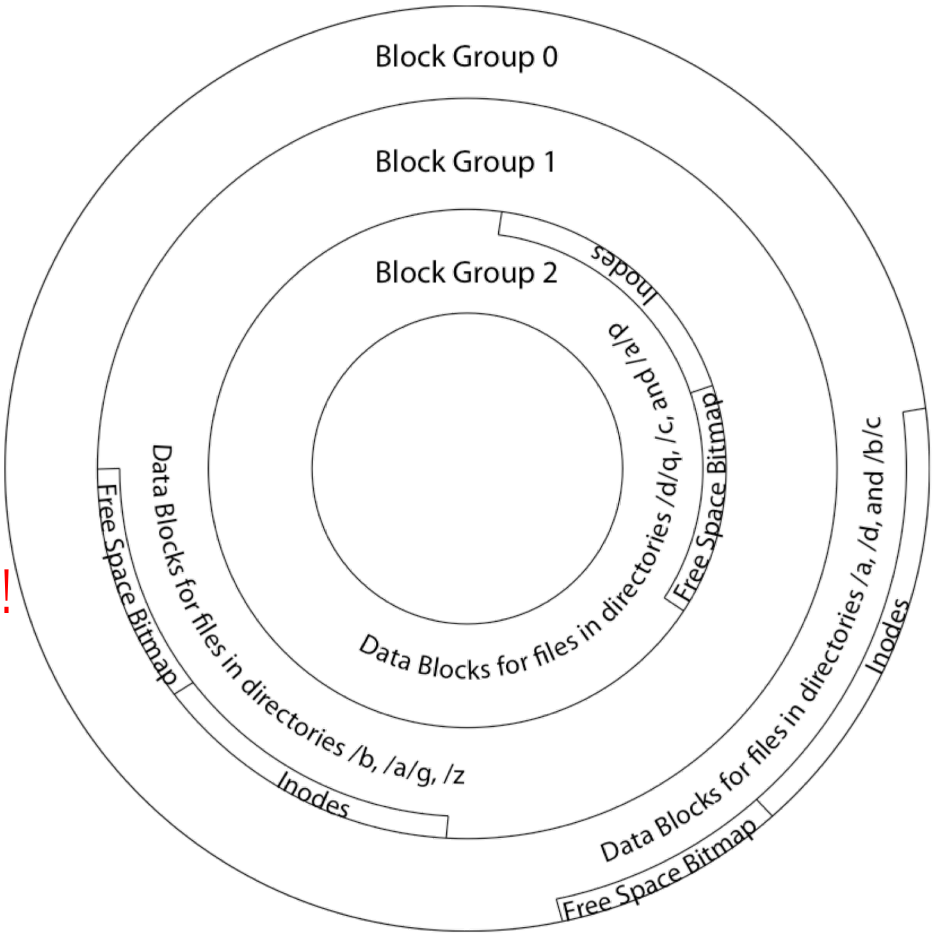
UNIX BSD 4.2 (1984)

- Problem: 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?
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other

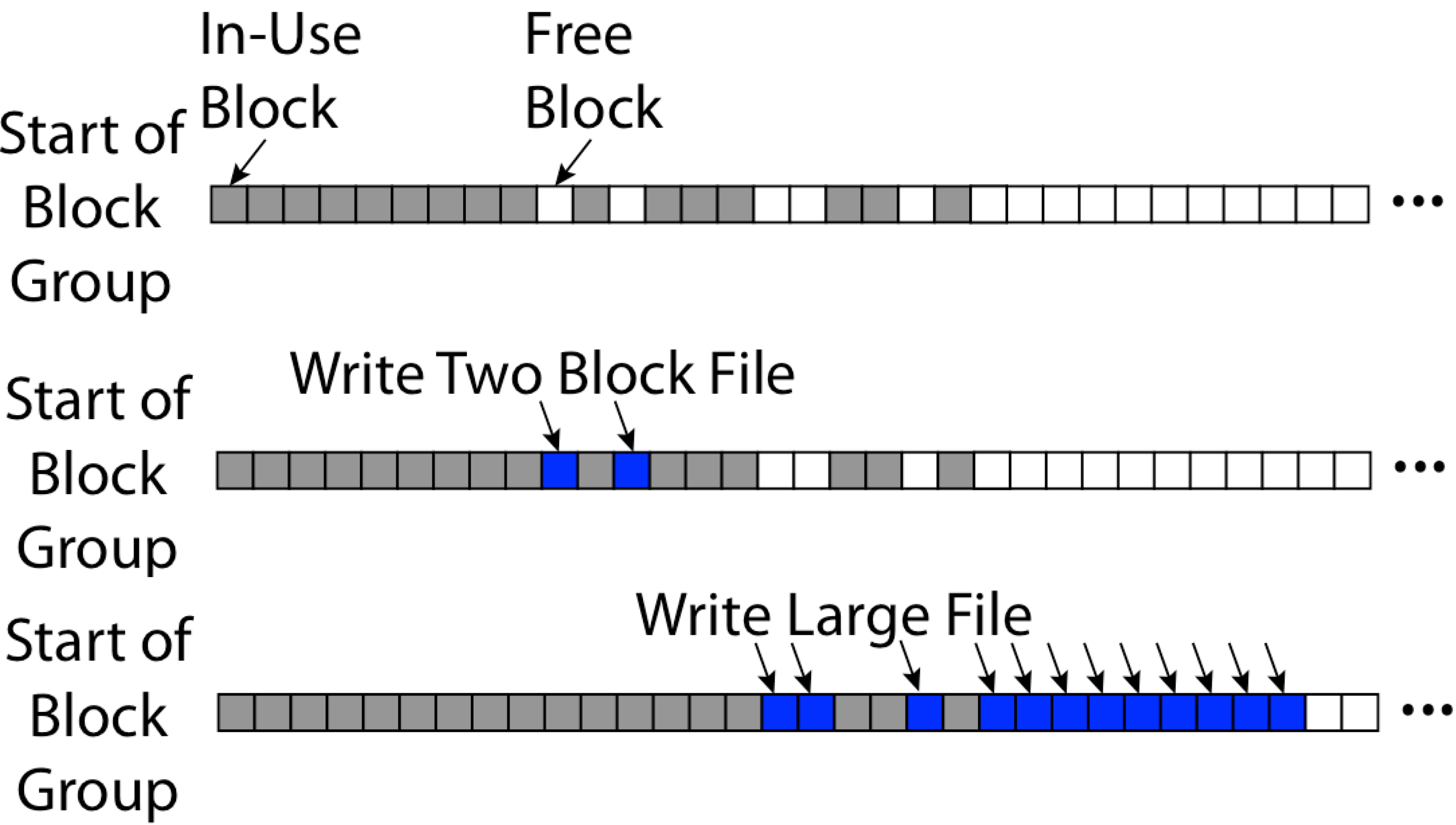
- Fast File System (FFS)
  - Allocation and placement policies for BSD 4.2

4.2 BSD Locality: Block Groups

- 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



UNIX 4.2 BSD FFS First Fit Block Allocation



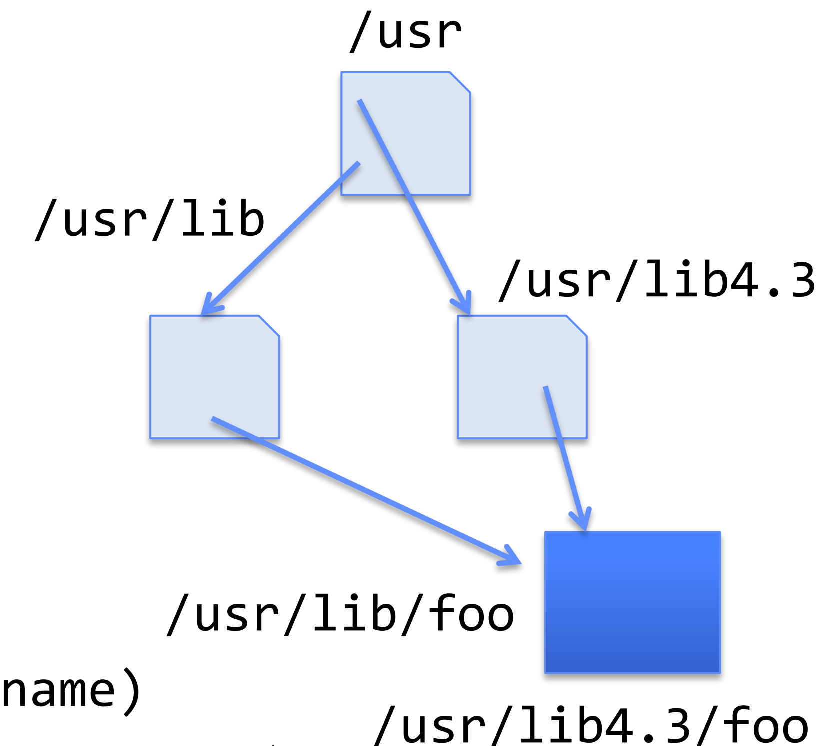
FFS Assessment

+ Efficient storage for large and small files
+ Locality for both file contents and metadata

- Inefficient for tiny files
  – E.g., a one-byte file requires 8KB of space on disk: inode and data block
- Inefficient encoding for contiguous ranges of blocks belonging to same file (e.g. blocks 4815 – 162342)

A bit more on directories

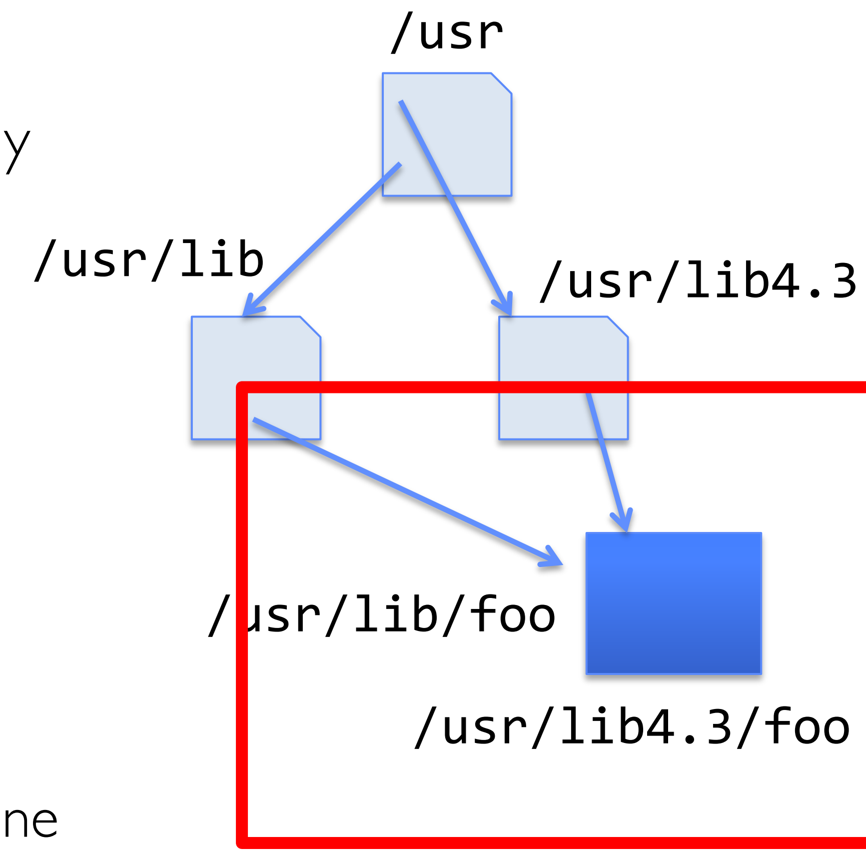
- Directories are specialized files
  - Contents: List of pairs
    <file name, file number>
- System calls to access directories
  - `open` / `creat` traverse the structure
  - `mkdir` /`rmdir` add/remove entries
  - `link` / unlink (`rm`)



- libc support
  - `DIR * opendir (const char *dirname)`
  - `struct dirent * readdir (DIR *dirstream)`
  - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`

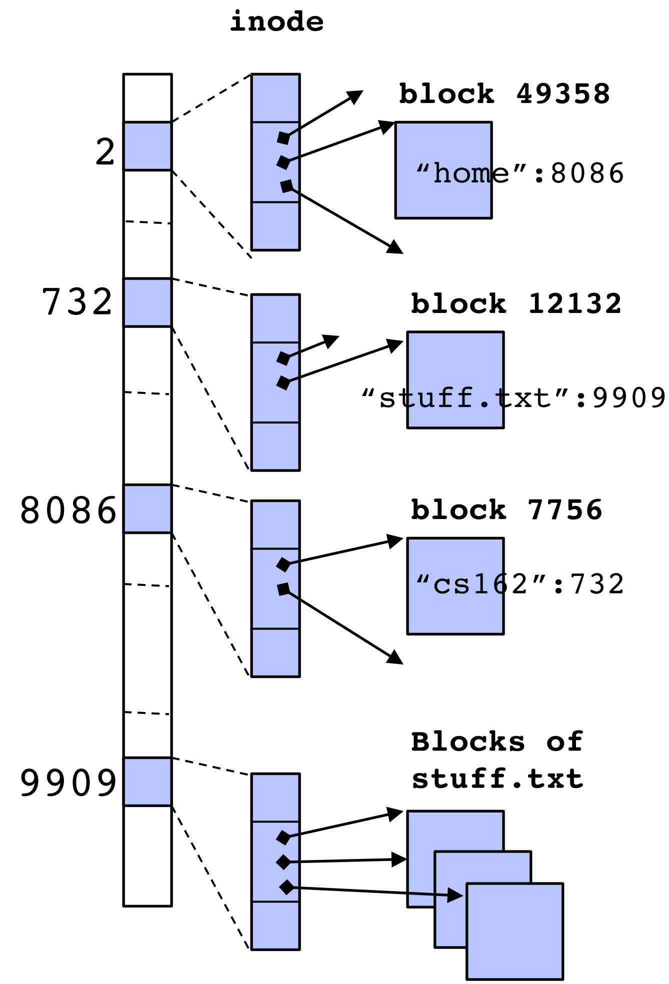
A Bit More on Directories

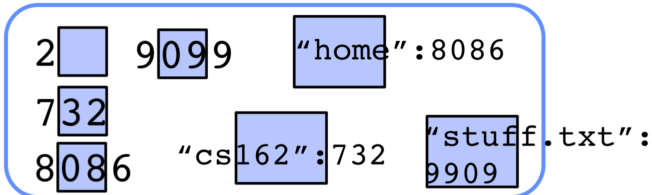
- Can we put the same file # in multiple directories?
- Unix: Yes! *Hard Link*
- Add first hard link when file is initially created
- Create extra links with the **`link`** syscall
- Remove links with **`unlink (rm)`**
- inode maintains reference count
  - When 0, free inode and blocks
- When can file be deleted?
  - Delete after the last reference is gone



Directories, Index, and Blocks, …

- What happens when we open **`/home/cs162/stuff.txt`**?
- / - inumber for root inode is configured into the kernel, say 2
  - Read inode 2 from its position in incode array on disk
  - Extract the direct and indirect block pointers
  - Determine block that holds the 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 descriptor to refer to this inode so reads / write can access the data blocks referenced by its direct and indirect pointers





Soft Links (Symbolic Links)

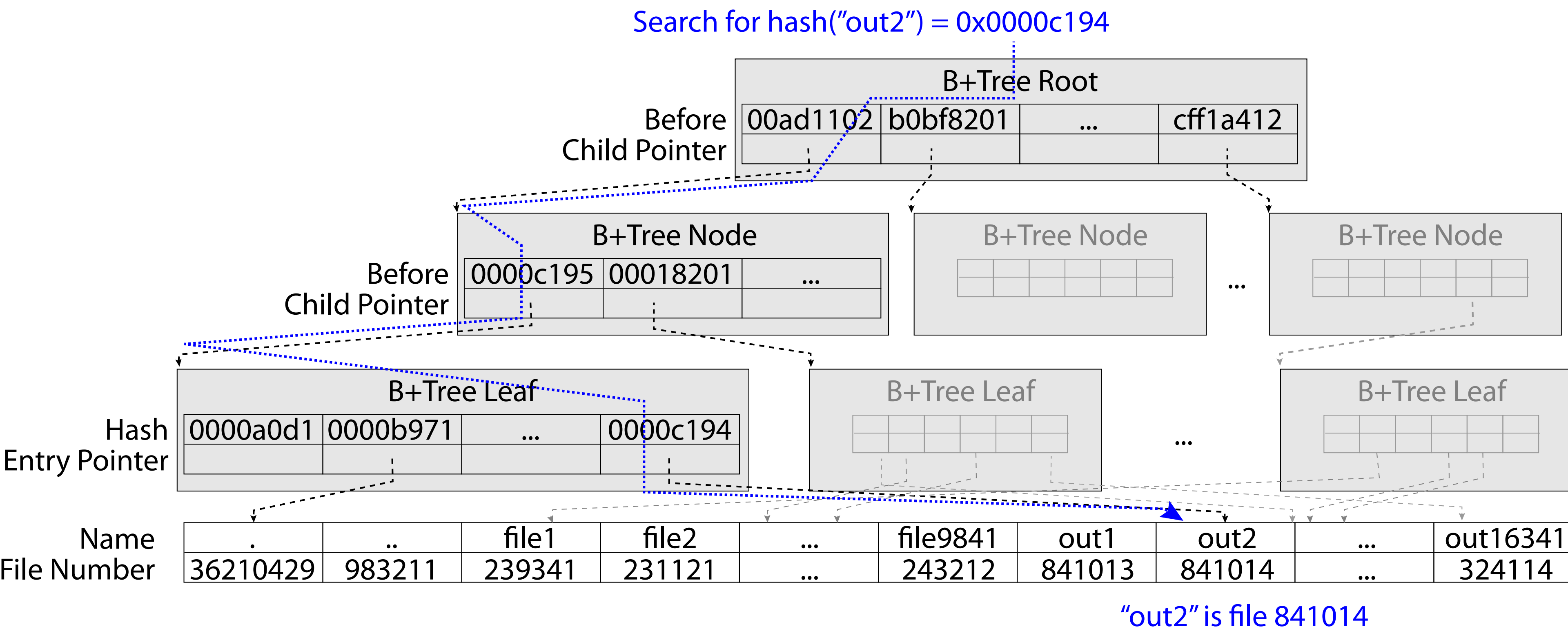
- Normal directory entry: <file name, file #>
- Symbolic link: <source file name, dest. file name>

- 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

Large Directories: B-Trees (dirhash)

in FreeBSD, NetBSD, OpenBSD



B Tree

- Balanced trees suitable for storing on disk
- Like balanced binary tree, but many more than 2 children
- Why? Remember we read/write in *blocks*
  - Make node roughly size of a block – manipulate in one disk operation
  - Sorted list of child nodes for each internal node of tree

Break

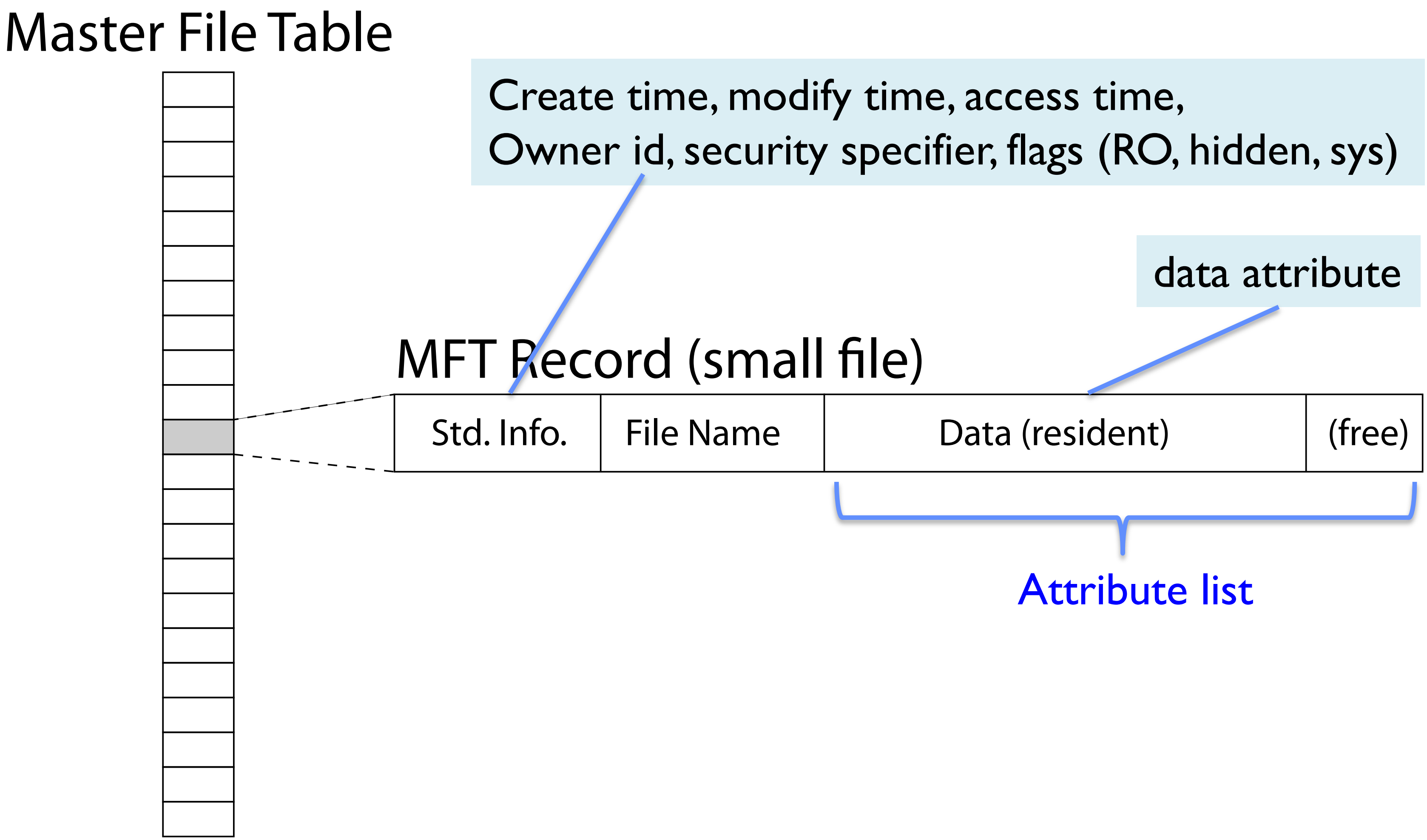
New Technology File System (NTFS)

- Default on modern Windows systems
- Instead of FAT or inode array: Master File Table
  - Max 1 KB size for each table entry
- Each entry in MFT contains metadata plus
  - 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

New Technology File System (NTFS)

- Default on modern Windows systems
- Instead of FAT or inode array: Master File Table
  - Max 1 KB size for each table entry
- Each entry in MFT contains metadata plus
  - 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

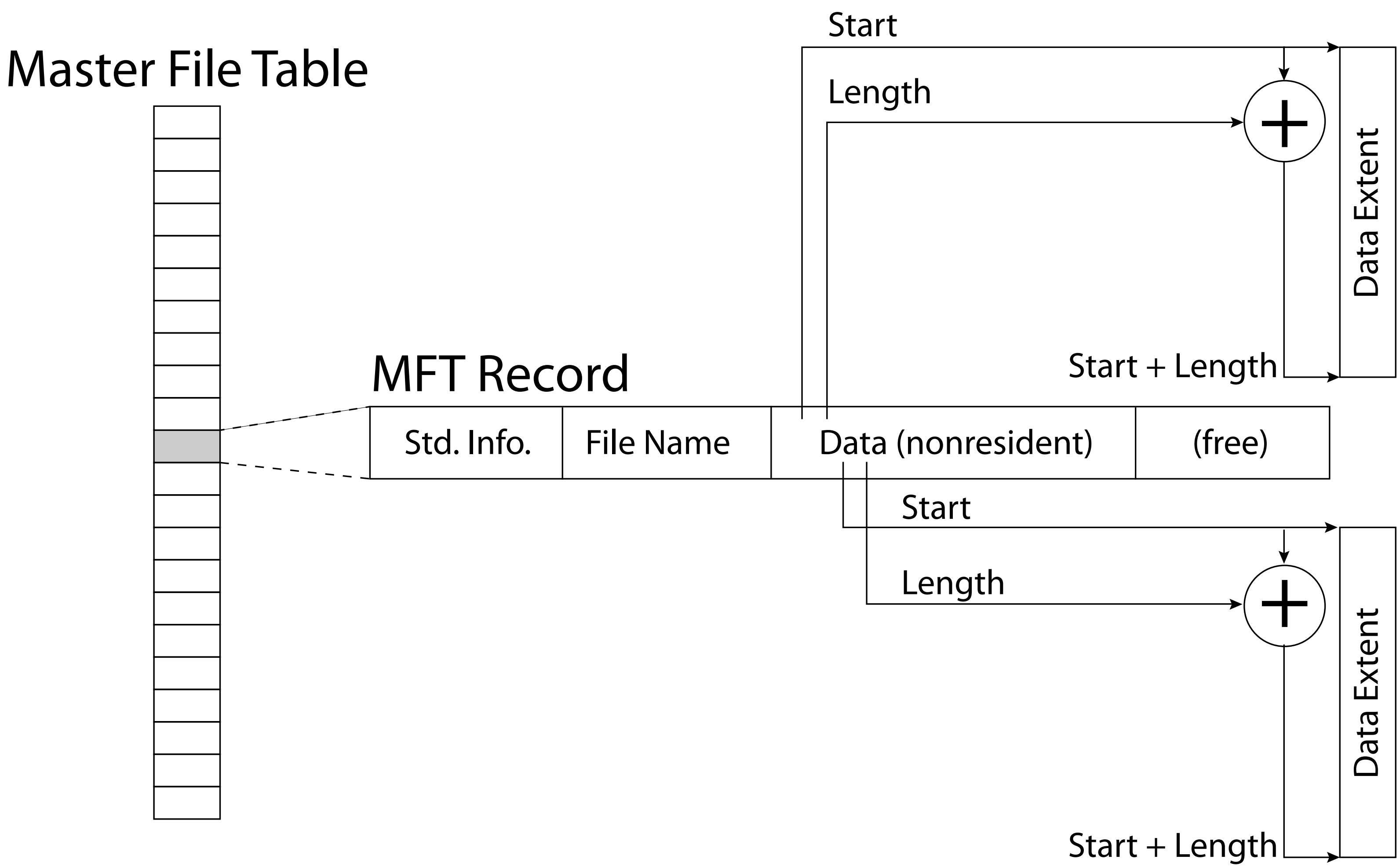
NTFS Small File



New Technology File System (NTFS)

- Default on modern Windows systems
- Instead of FAT or inode array: Master File Table
  - Max 1 KB size for each table entry
- Each entry in MFT contains metadata plus
  - 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 Medium File



Why Extents?

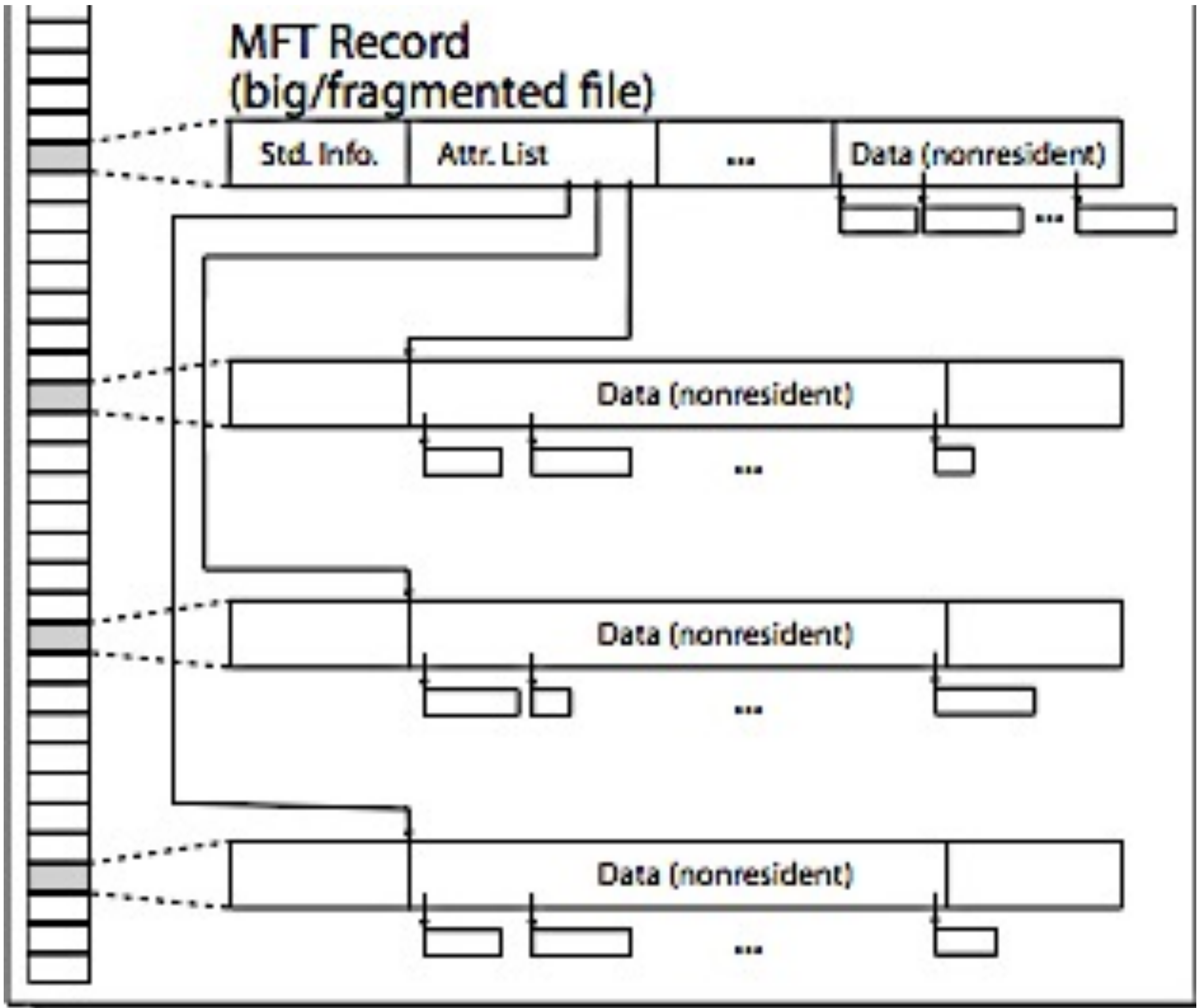
- FFS: List of *fixed size blocks*
- For larger files, we want their contents to be on contiguous blocks anyways

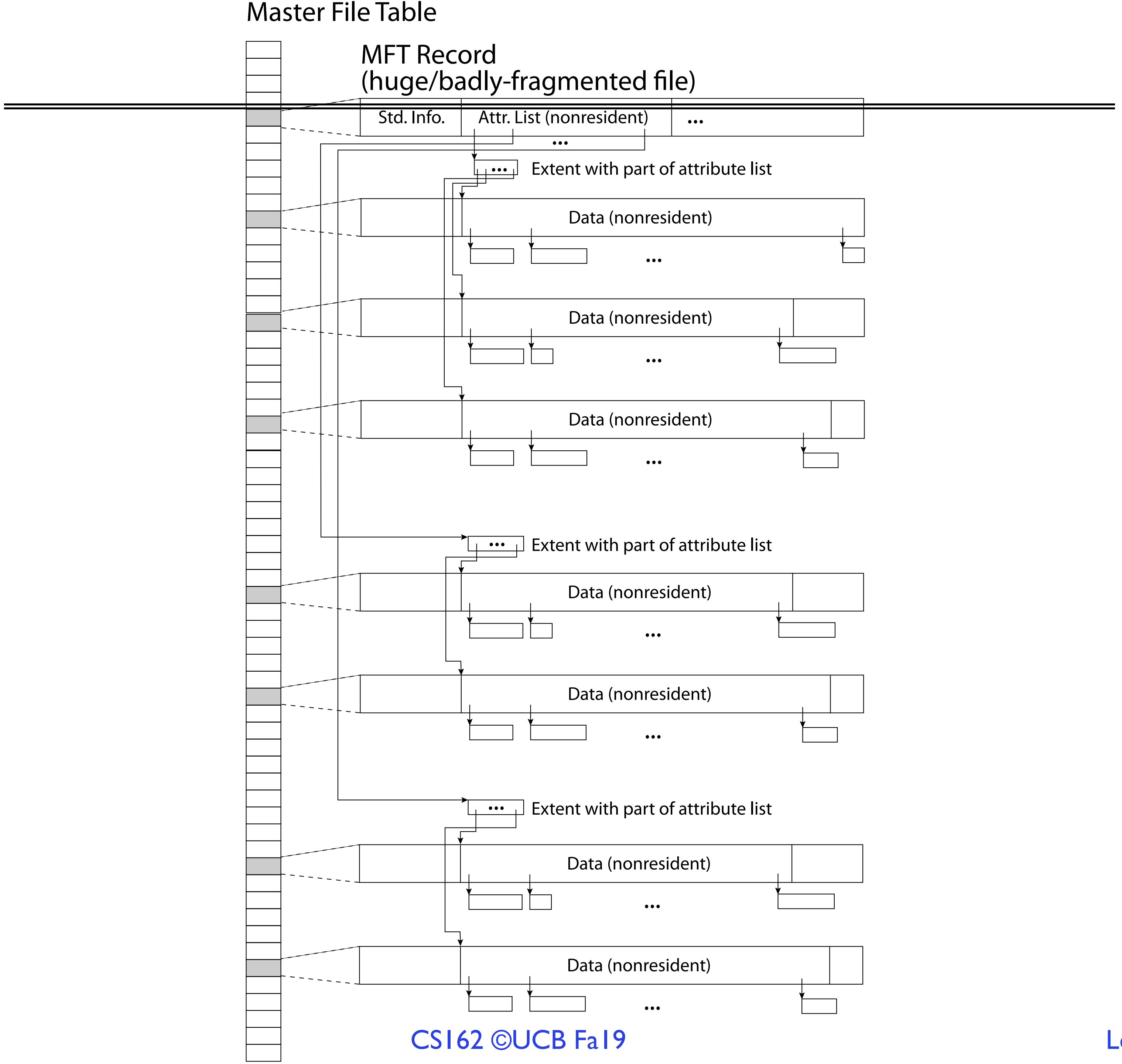
- Idea: Store starting block and number of subsequent contiguous blocks
- File made of 1000 sequential blocks
  – Extents: Just one metadata entry
  – Blocks: 1000 entries (plus indirect pointer!)

New Technology File System (NTFS)

- Default on modern Windows systems
- Instead of FAT or inode array: Master File Table
  - Max 1 KB size for each table entry
- Each entry in MFT contains metadata plus
  - 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 Multiple Indirect Blocks





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 Directories

- Directories implemented as B Trees
- 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

Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

How to Make File System Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  – Can allow recovery of data from small media defects

- Make sure writes survive in short term
  – Either abandon delayed writes or
  – Use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache

- Make sure that data survives in long term
  – Need to replicate! More than one copy of data!
  – Important element: independence of failure
    » Could put copies on one disk, but if disk head fails…
    » Could put copies on different disks, but if server fails…
    » Could put copies on different servers, but if building is struck by lightning….
    » Could put copies on servers in different continents…

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
- 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 System Summary (2/2)

- 4.2 BSD Multilevel index 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
- File layout driven by freespace management
  - Integrate freespace, inode table, file blocks and dirs into block group
- Reliability, Availability & Durability