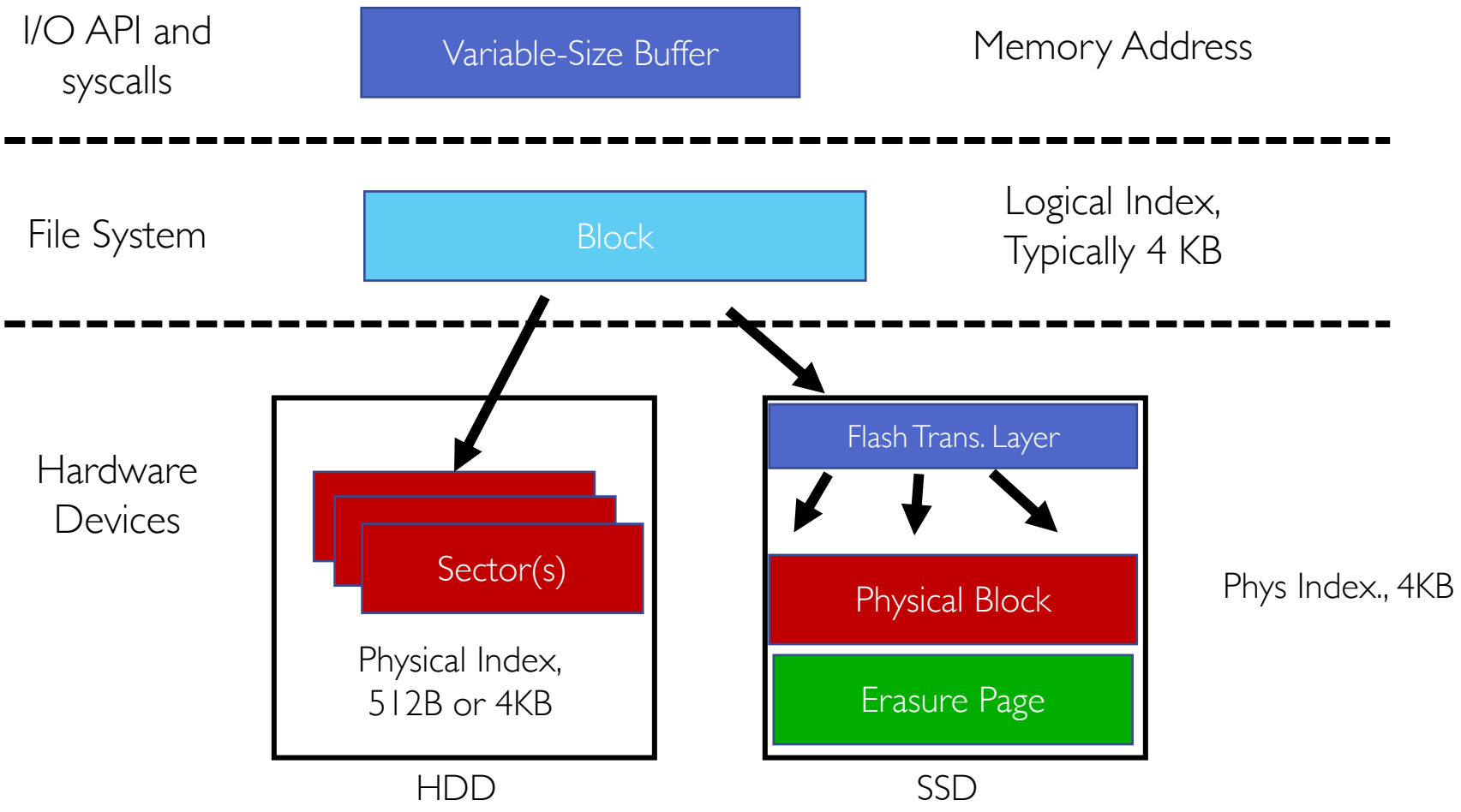
SE350: Operating Systems

Lecture 14: File Systems

Outline

- File Systems
- File Allocation Table (FAT)
- Unix Fast File System (FSS)
- New Technology File System (NTFS)

From Storage to File Systems



Building File Systems

- File system is layer of OS that transforms block interface of disks (or other block devices) into files, directories, etc.

- Goal: Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with
  - Naming: Interface to find files by name, not by blocks
  - Disk management: Collect disk blocks into files
  - Protection: Keep data secure and isolated
  - Reliability/Durability: Keep files durable despite crashes, failures, attacks, etc.

Recall: User vs. System View of Files

- User's view
  - Durable data structures
- System's view (system-call interface)
  - Collection of bytes (UNIX)
  - Doesn't matter what kind of data structures you want to store on disk!
- System's view (inside OS)
  - Collection of blocks (block: Logical transfer unit; sector: Physical transfer unit)
  - Block size ≥ sector size (4 KB in Unix)

Recall: From User to System View

- What happens if user asks for bytes 2 to 12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about writing bytes 2 to 12?
  - Fetch block, modify relevant portions, write out block
- Everything inside file system is in whole size blocks
  - Actual disk I/O happens in blocks
  - E.g., even **`getc()`** and **`putc()`** need to be translated and buffered in blocks

What Do File Systems Need to Do?

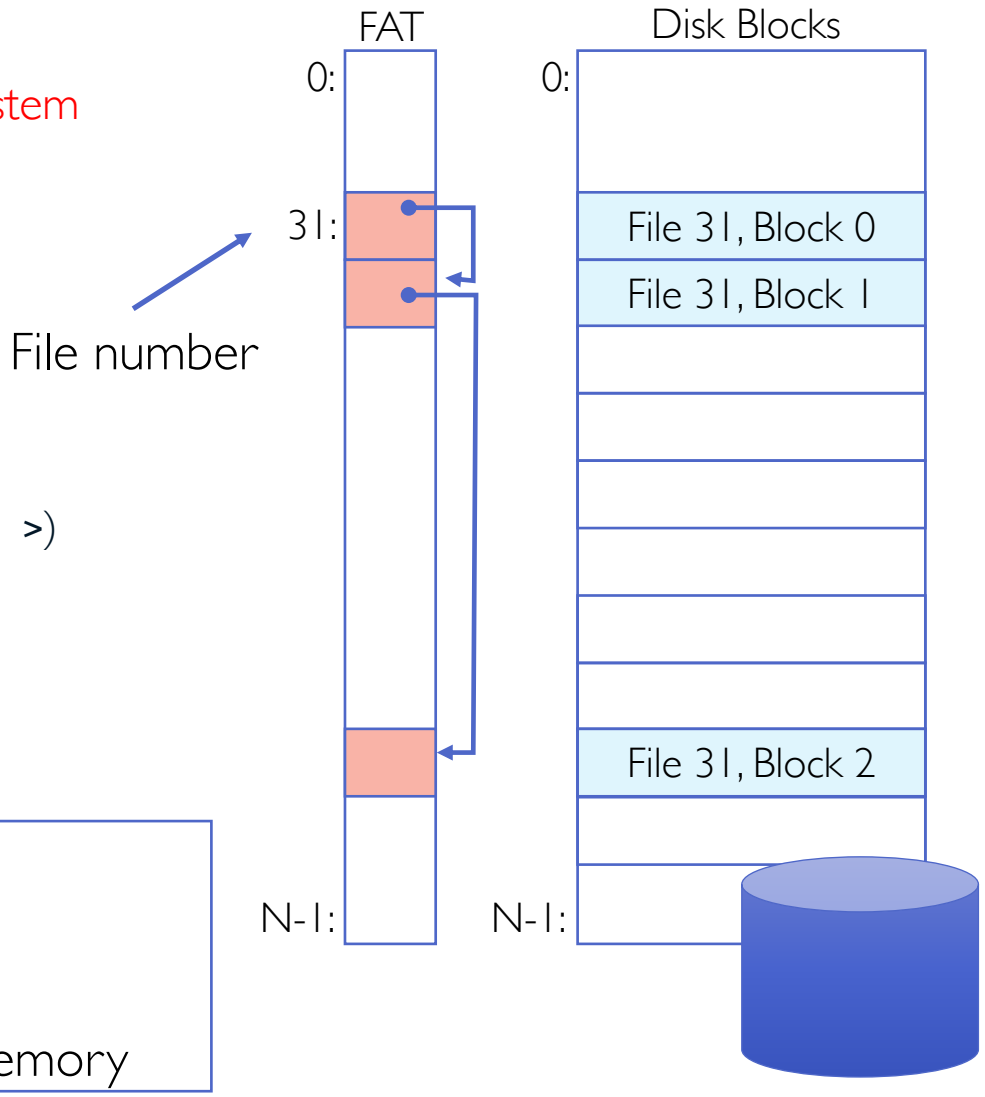
- Track free disk blocks
  - Need to know where to put newly written data
- Track which blocks contain data for which files
  - Need to know where to read a file from
- Track files in directory
  - Find list of file's blocks given its name
- Where do we maintain all of this?
  - Somewhere on disk
  - For durability, state of file system must be meaningful upon shutdown
    - This obviously isn't always the case…

File

- Named permanent storage, containing
  - Data: Blocks on disk somewhere
  - Metadata (attributes)
    - Owner, size, last opened, …
    - Access rights (i.e., R, W, X)
      - Owner, group, other (Unix)
      - Access control list (Windows)

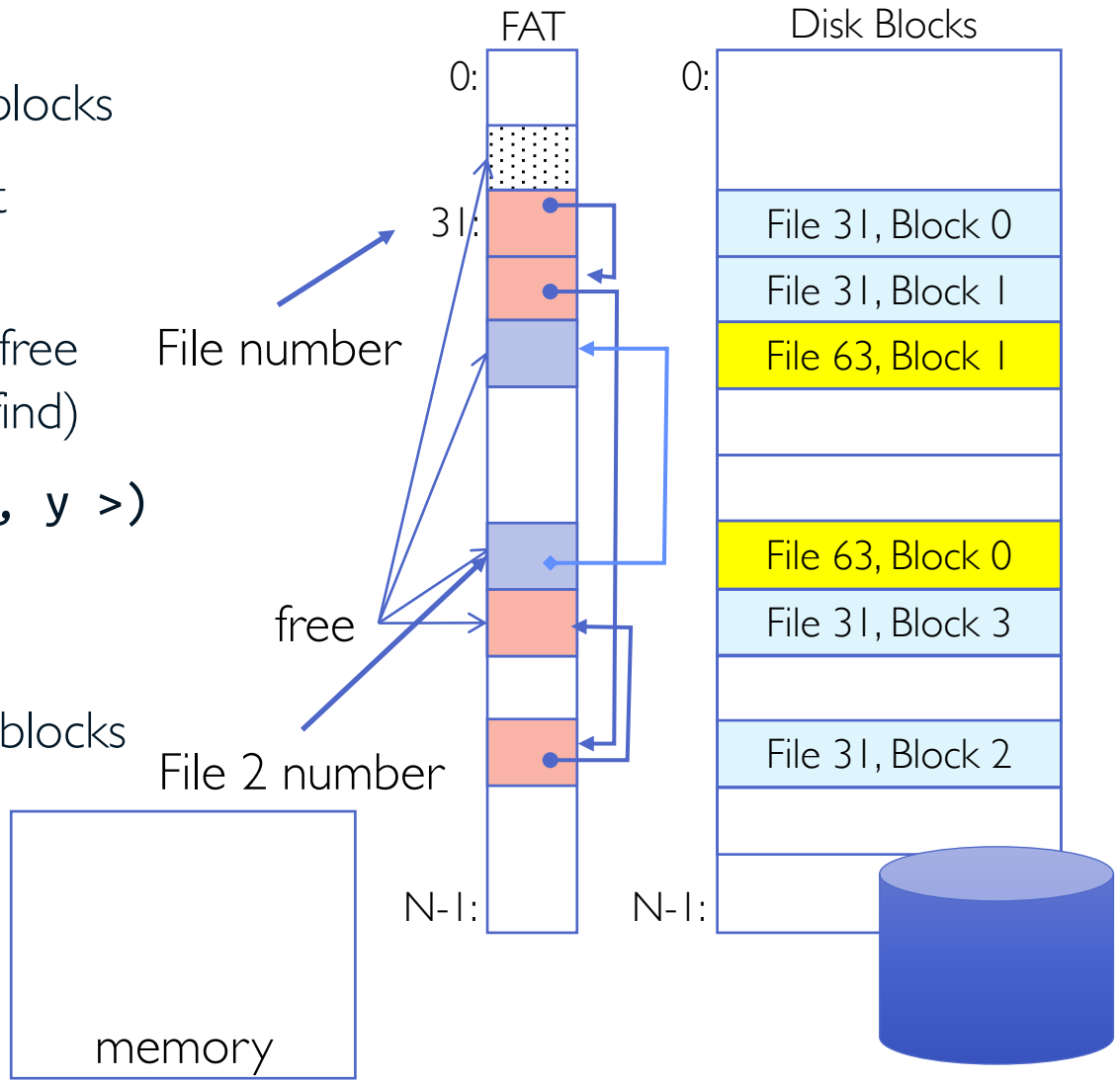
Our First Filesystem: FAT (File Allocation Table)

- MS-DOS, 1977, still widely used file system (thumb drives, boot, …)
- Assume (for now) we have a way to translate a path to a "file number"
  - I.e., directory structure
- Disk storage is collection of blocks
  - Just hold file data (offset `o = < B, x >`)
  - E.g., `file_read(31, < 2, x >)`
- Index into FAT with file number
- Follow linked list to block
- Read block from disk into memory



FAT Properties

- FAT is linked list `1-1` with blocks
- File number is index of root of block list for the file
- Unused blocks are marked free (no ordering, must scan to find)
- E.g., `file_write(31, < 3, y >)`
  - Grab free block
  - Linking them into file
- Grow file by allocating free blocks and linking them in
- E.g., create file, write, write



FAT Assessment

- Where is FAT stored?
  - On disk, on boot cache in memory, second (backup) copy on disk
- What happens when you format disk?
  - Zero all blocks, mark FAT entries "free"
- What happens when you quick format disk?
  - Mark all entries in FAT as free
- Pros: (FAT is simple!)
  - Easy to find free block
  - Easy to append to files
  - Easy to delete files

FAT Assessment (cont.)

- Cons:
  - Random access is very slow (traversing linked-list)
    - Takes long time to find random block in large file
    - Sequential accesses have lower overhead
  - Fragmentation and poor locality
    - File blocks for a given file may be scattered
    - Files in the same directory may be scattered
    - Problem becomes worse as disk fills
    - *Defragmentation* could be used to improve performance
  - Security vulnerabilities
    - FAT does not support access rights
    - FAT has no header in file blocks

Unix Fast File System (FFS)

- Inode structure is used to track file blocks
  - Originally, inode appeared in *Berkeley Standard Distribution (BSD) 4.1*
  - Similar structure for Linux Ext2/3
- File number is index into inode array
- Inode stores metadata associated with each file
- Inode has multi-level index structure
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks
- Free space is managed using bitmap with one bit per storage block

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

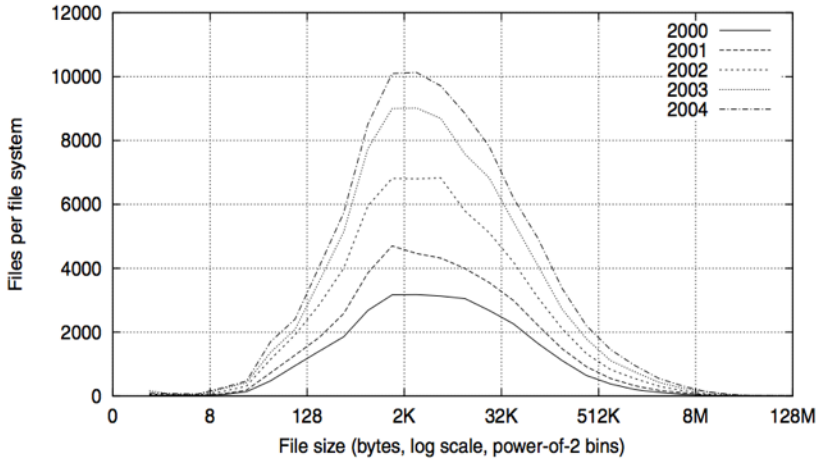


Fig. 2. Histograms of files by size.

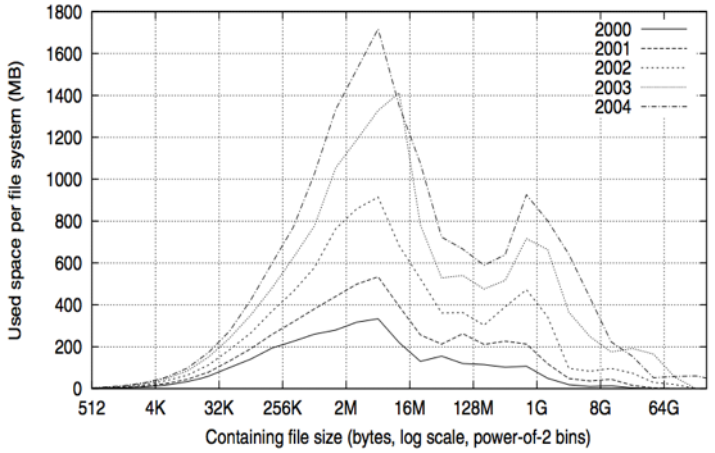


Fig. 4. Histograms of bytes by containing file size.

Characteristics of Files (cont.)

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

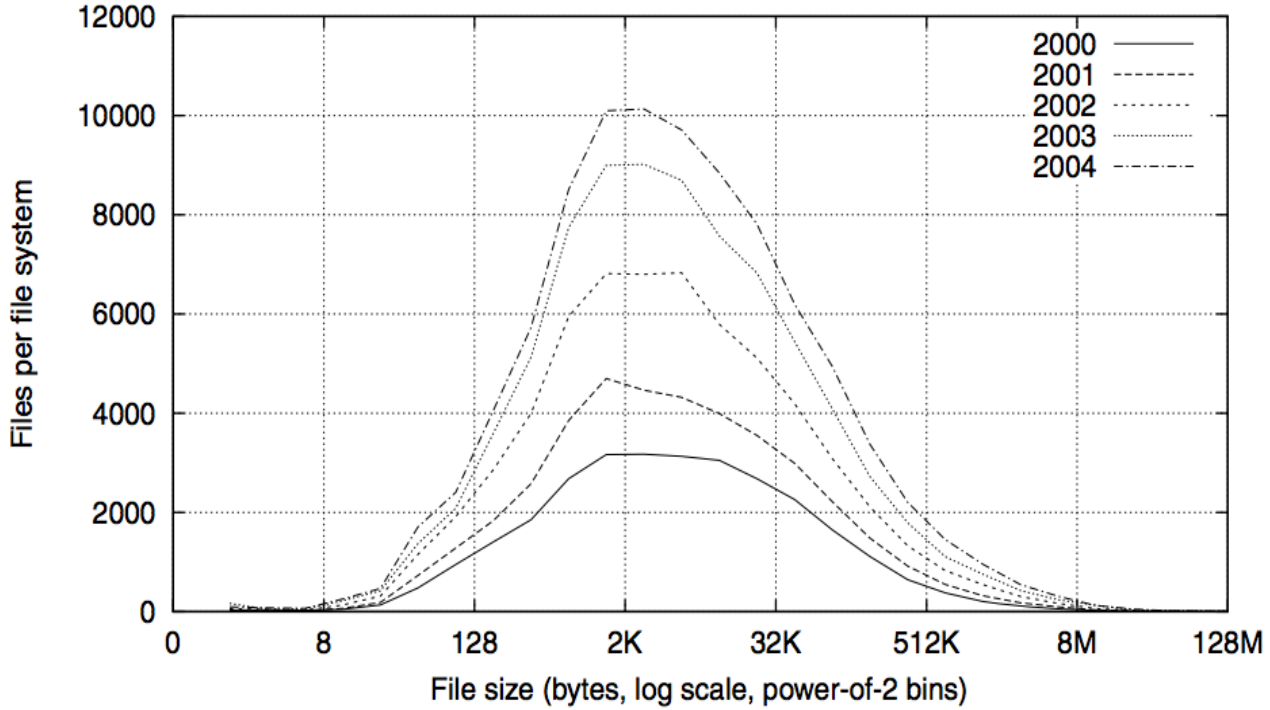


Fig. 2. Histograms of files by size.

Characteristics of Files (cont.)

- Most of disk space is occupied by rare big ones

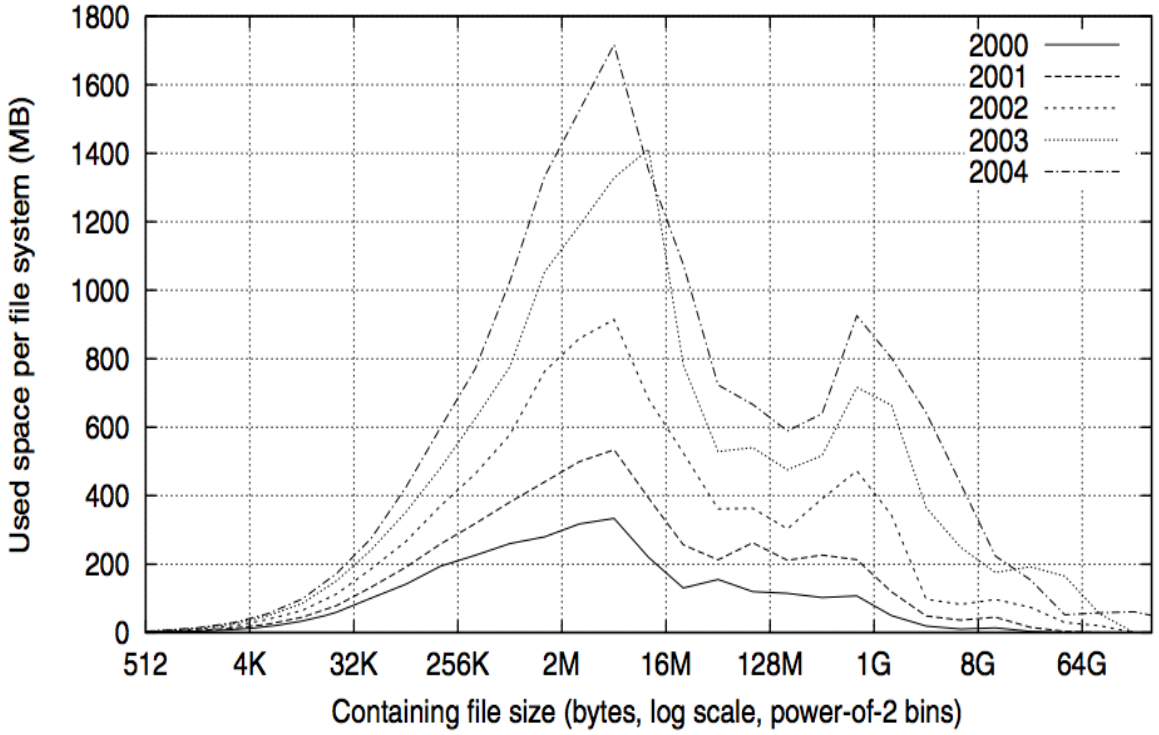
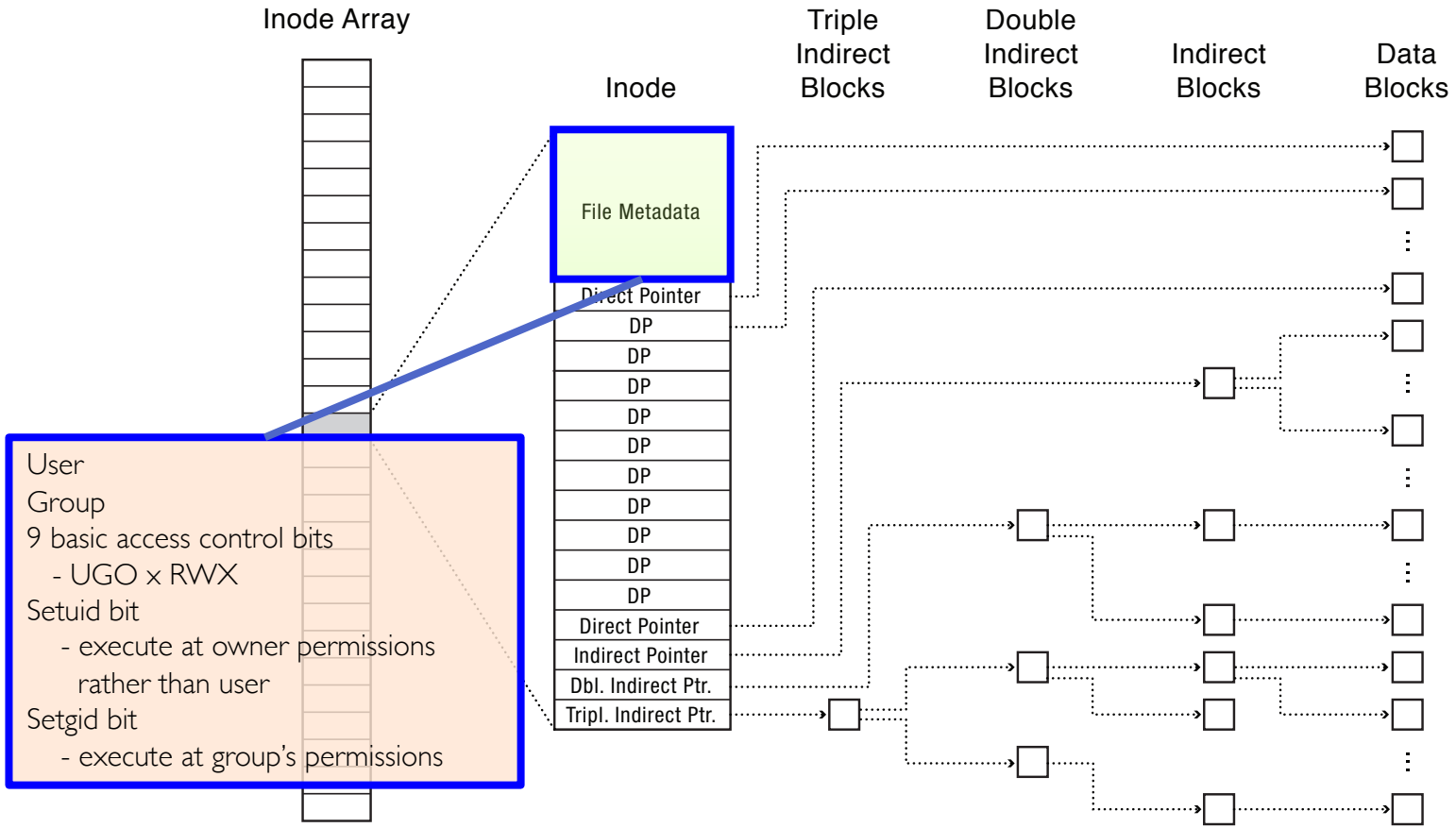
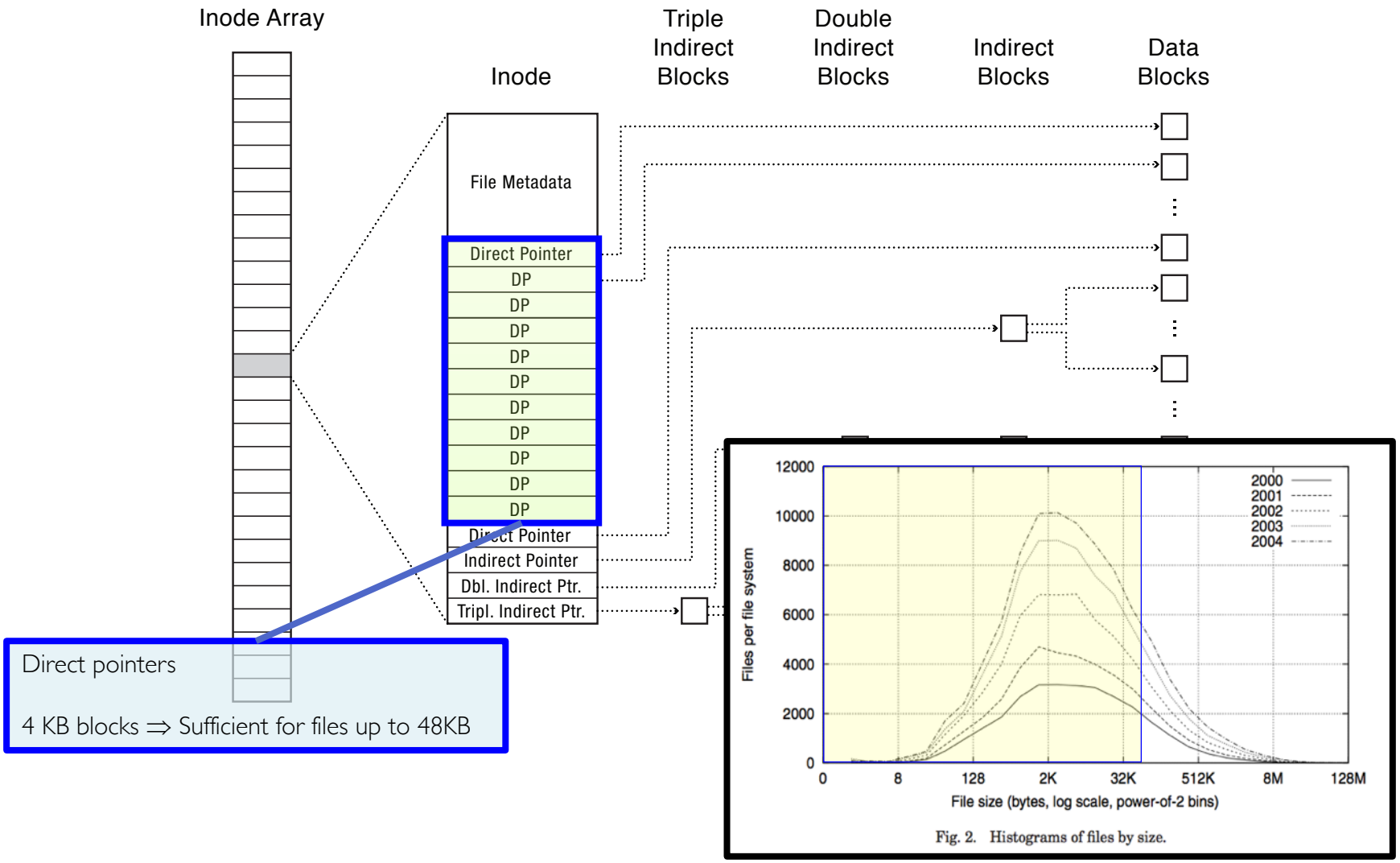


Fig. 4. Histograms of bytes by containing file size.

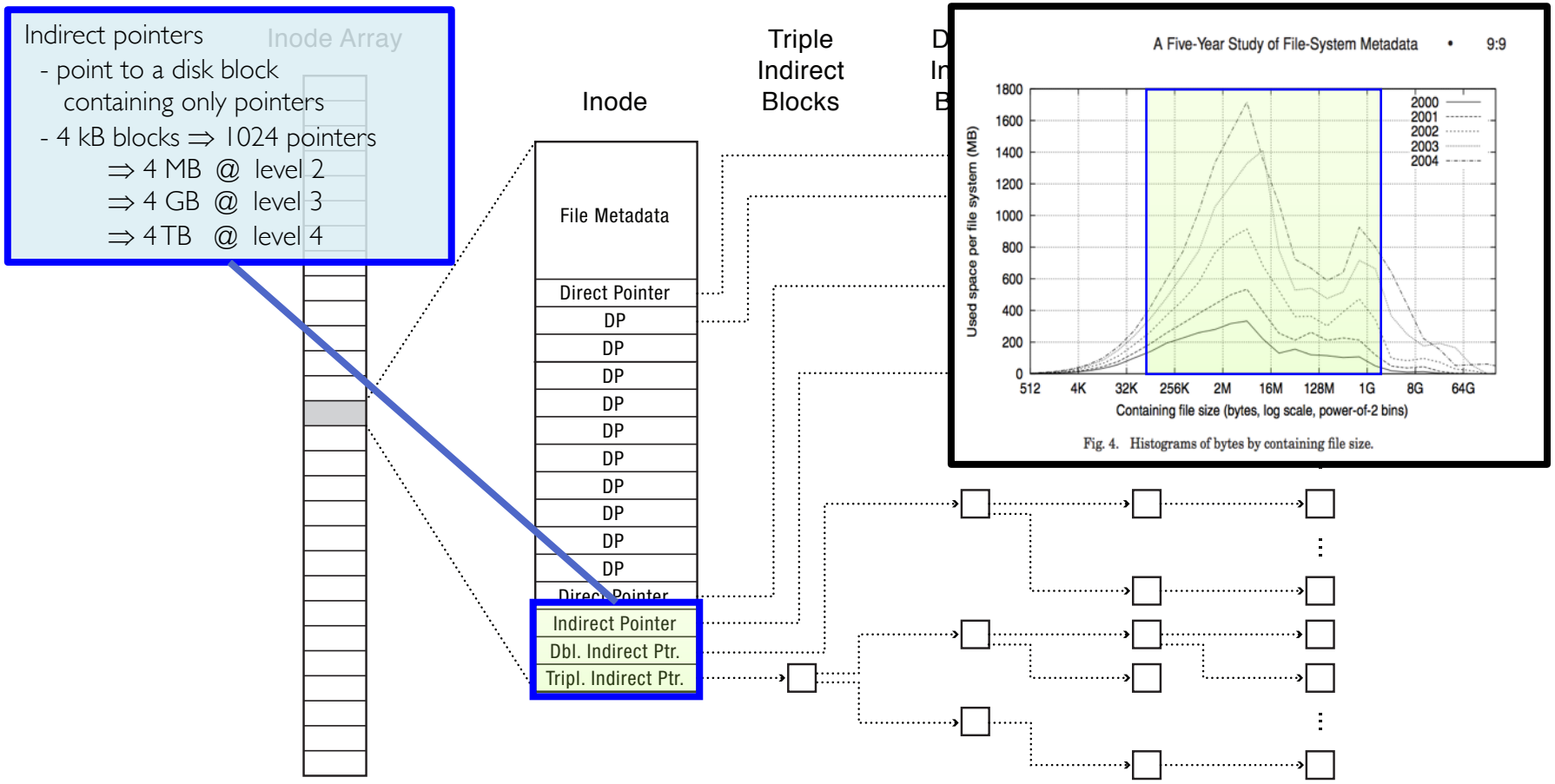
Inode Structure



Inode Structure (cont.)



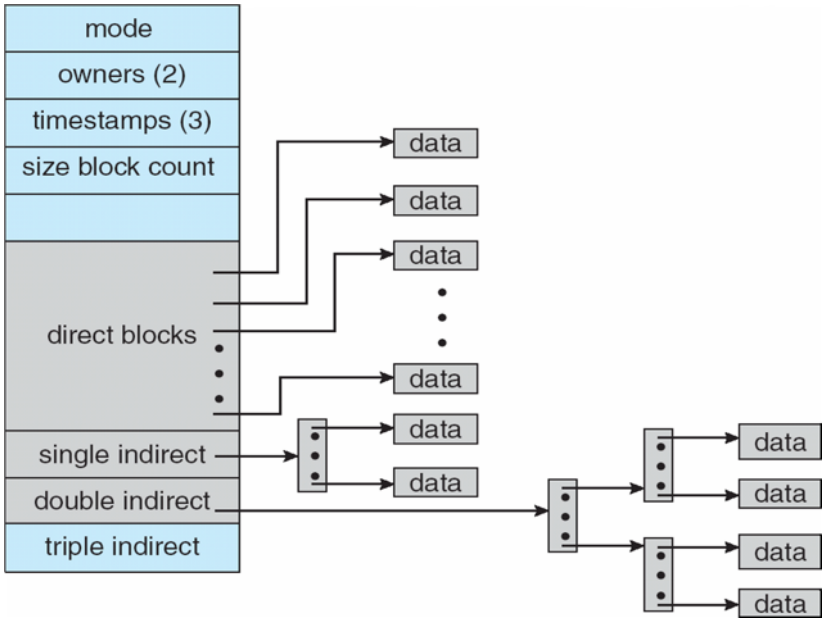
Inode Structure (cont.)



Inode Details

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4 KB blocks ⇒ max size of 48 KB
- Indirect block pointer
  - Pointer to disk block of data pointers
  - 4 KB block size and 4-byte block pointer ⇒ 1 K direct block pointers
  - 4 MB (+ 48 KB)
- Doubly indirect block pointer
  - Doubly indirect block ⇒ 1 K indirect blocks
  - 4 GB (+ 4 MB + 48 KB)
- Triply indirect block pointer
  - Triply indirect block ⇒ 1 K doubly indirect blocks
  - 4 TB (+ 4 GB + 4 MB + 48 KB)

Multilevel Indexed Files



- Small files: shallow tree
  - Efficient storage for small files
- Large files: deep tree
  - Efficient lookup for random access in large files
- Sparse files: only fill pointers if needed

Multilevel Indexed Files Example

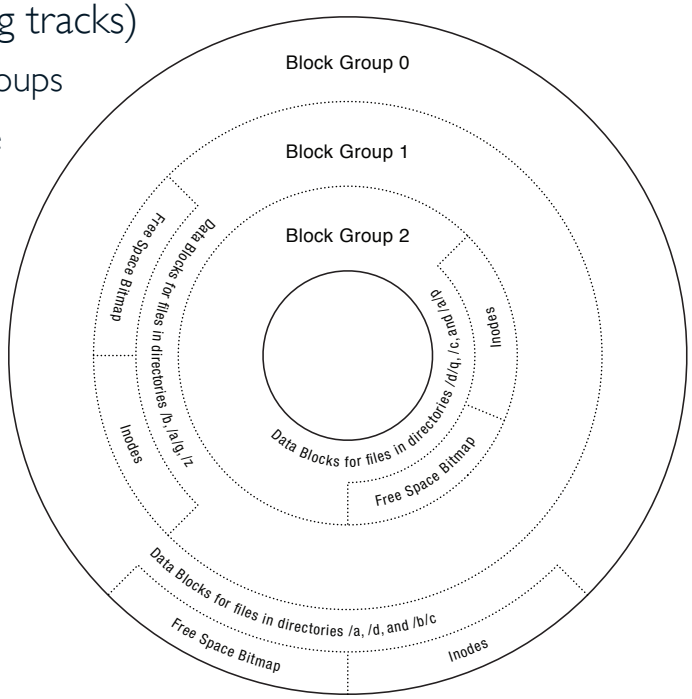
- How many accesses for block #5 (assuming file header accessed on open)?
  - One: One for data
- How many accesses for block #23?
  - Two: One for indirect block, one for data
- How many accesses for lock #340?
  - Three: double indirect block, direct block, and data
- …

Where Are Inodes Stored?

- In early FFS, inode array was stored at fixed location in outermost cylinders
  - At formatting time, fixed number of inodes are created
  - Each is given a unique number, called an inumber
  - To read small files, disk head had to seek to get header, seek back to data

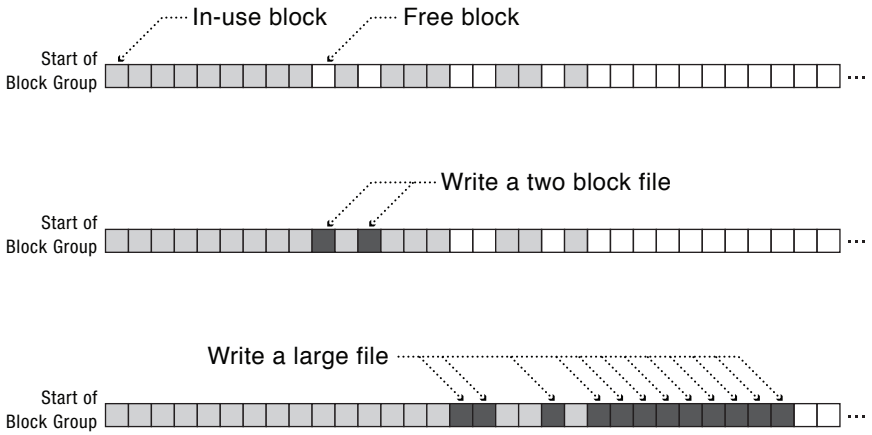
- In later versions, disk is divided into groups (neighboring tracks)
  - Data blocks, inodes, and bitmaps are scattered across groups
  - Avoid seeks between user data and file system structure
  - Put directory and its files in the same block group

- Pros?
  - For small directories, we can fit data blocks and inodes in same cylinder ⇒ no seeks!
  - Reliability: whatever happens to disk, OS can find most of files (even if directories disconnected)



Data Block Placement: First-Free Heuristic

- To expand file, use first free blocks in group using the group's bitmap



- This might scatter sequential writes into holes near start of group
- But it leads to contiguous free space at the end of group
  - Reduces fragmentation
  - This is less effective if disk is almost full
    - There will be only scattered holes in each group
  - To keep this effective, FSS keeps 10% or more of disk free!
    - Reserve space for super user's processes

FSS Analysis

- 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 B file requires both inode and data block)
  - Inefficient encoding when file is mostly contiguous on disk
  - Need to reserve 10-20% of free space to prevent fragmentation

Directories of Files!

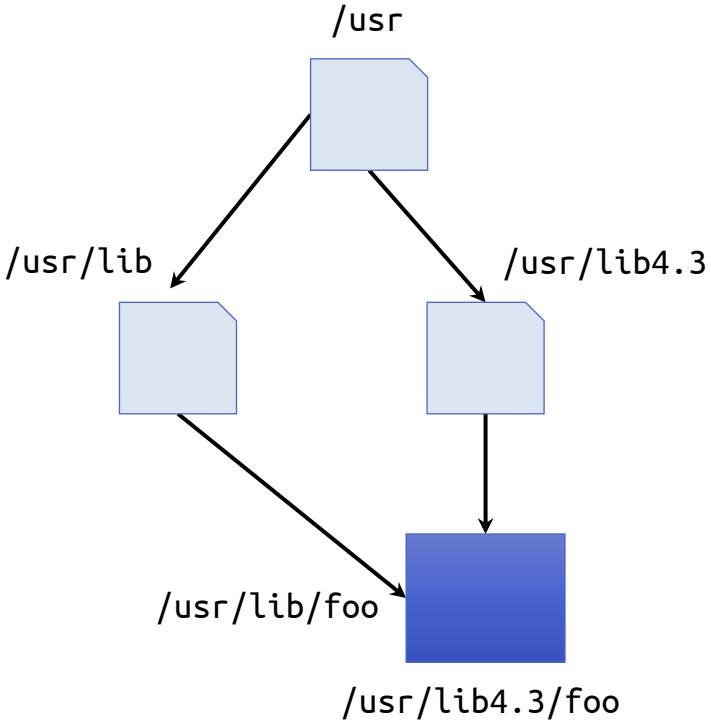
howtogeek.com

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
  - Softlinks (aliases) are another name for an entry

Directory Details

- Directories are stored in files
  - Can be read and written to
- System calls to access directories
  - `open/create` to traverse the structure
  - `mkdir/rmdir` to add/remove entries
  - `link/unlink(rm)` to link/unlink existing file to directory
    - Not in FAT !
    - Forms a DAG
- When can file be deleted?
  - Maintain ref-count of links to each file
  - Delete after last reference is gone
- `Libc` support
  - `DIR * opendir (const char *dirname)`
  - `struct dirent * readdir (DIR *dirstream)`
  - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`

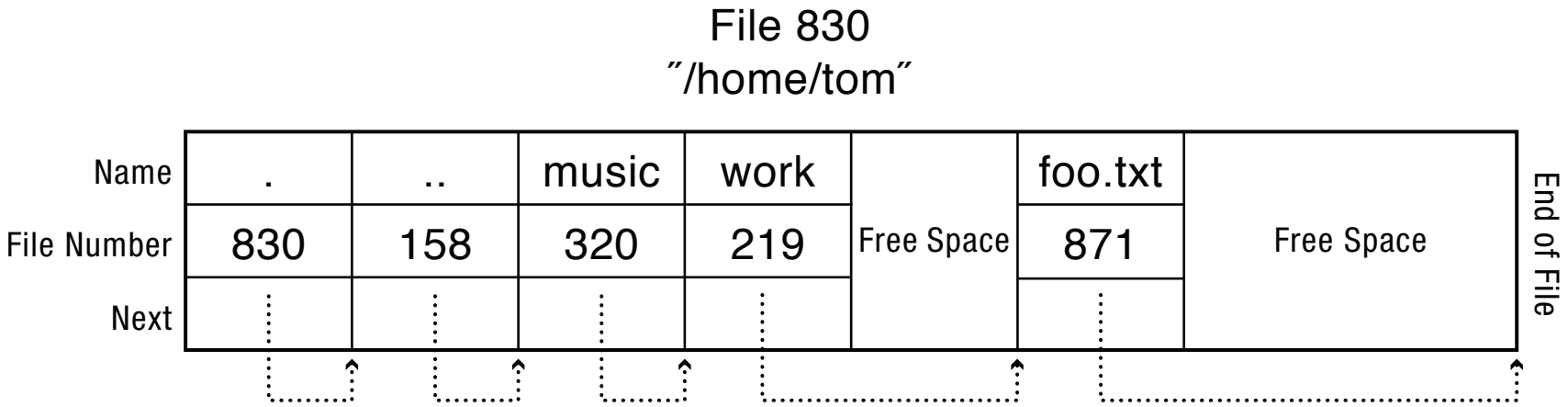


Links

- Hard link
  - E.g., **`ln /path/to/original/file /path/to/link`** (in Unix)
  - Sets another directory entry to contain file number for original file
  - Creates another name (path) for original file
  - Each directory entry is "first class"
  - If original file is deleted, the other one is still valid

- Soft link or *symbolic link* or *shortcut*
  - E.g., **`ln -s /path/to/original/file /path/to/link`** (in Unix)
  - Directory entry contains path and name of linked file
  - Map one name to another name
  - If original file is deleted, the soft link will point to nothing

Directory Structure

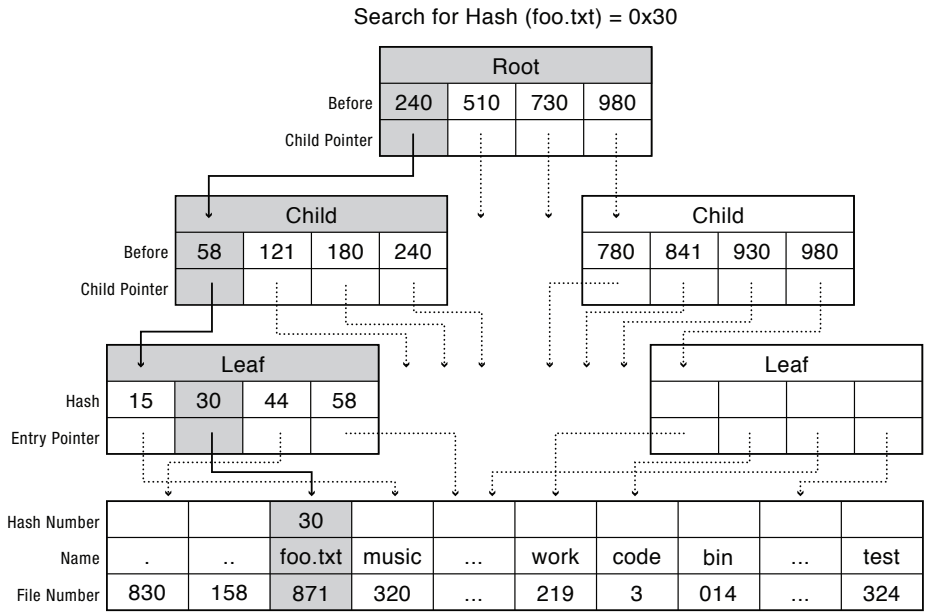


- Each directory is linked-list of entries
- Each entry contains `<file_name: file_number>` mapping
- In FAT, file attributes are kept in directory (!)

Directory Structure Example

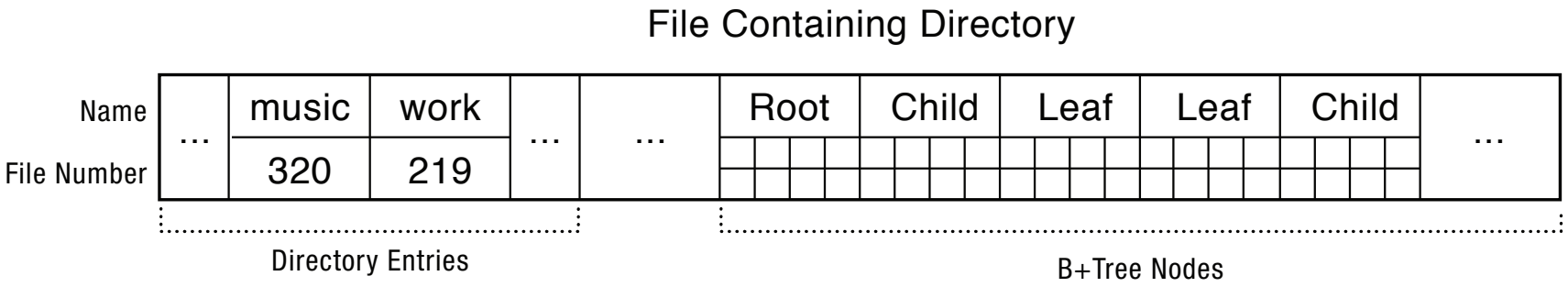
- How many disk accesses to resolve "`/usr/lib/foo`"?
  - Read in file header for root (fixed spot on disk)
    - Search linearly for "`usr`"– ok since directories typically very small
  - Read in file header for "`usr`"
    - Search for "`lib`"
  - Read in file header for "`lib`"
    - Search for "`foo`"
  - Read in file header for "`foo`"
- 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 = "/usr/lib"` can resolve "`foo`")

B+Tree Directory Layout: Modern File Systems



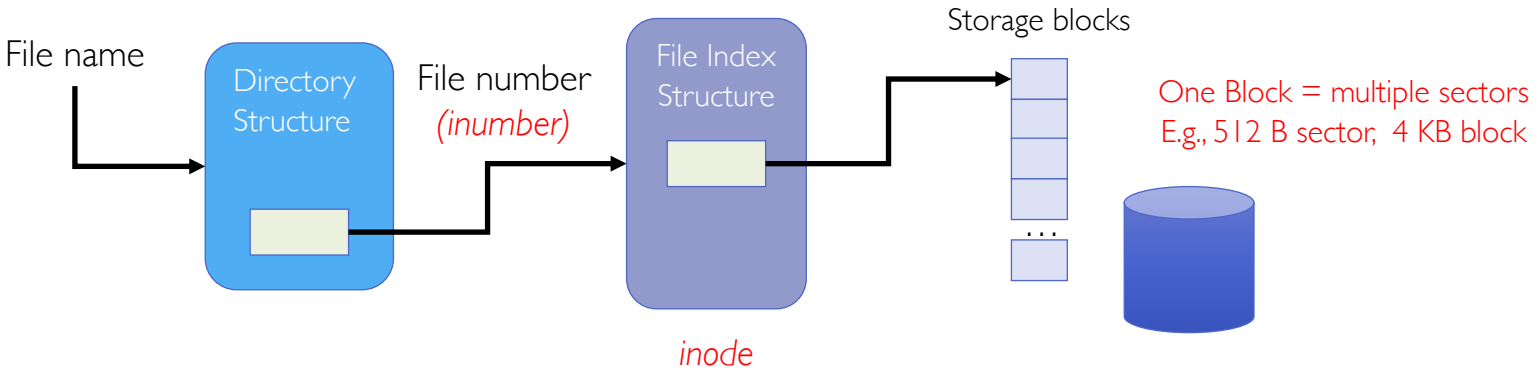
- B+tree is indexed by hash of file's name
- Internal nodes contain array of sorted hash keys, each pointing to child node
- Child node's keys are smaller than parent's key entry but larger than parent's previous keys
- File system searches node for first entry with key larger than target
- File number at leaf nodes points to target directory entry

B+Tree Directory Layout: Modern File Systems (cont.)



- Directory entries are usually stored in first part of directory file
- B+tree's root is at well-known offset within file
- Fixed-size internal and leaf nodes are stored after root node
- Variable-size directory entries are stored at start of directory file
- Each tree node includes pointers to where in the file its children are stored

Putting it Together: Unix File System API



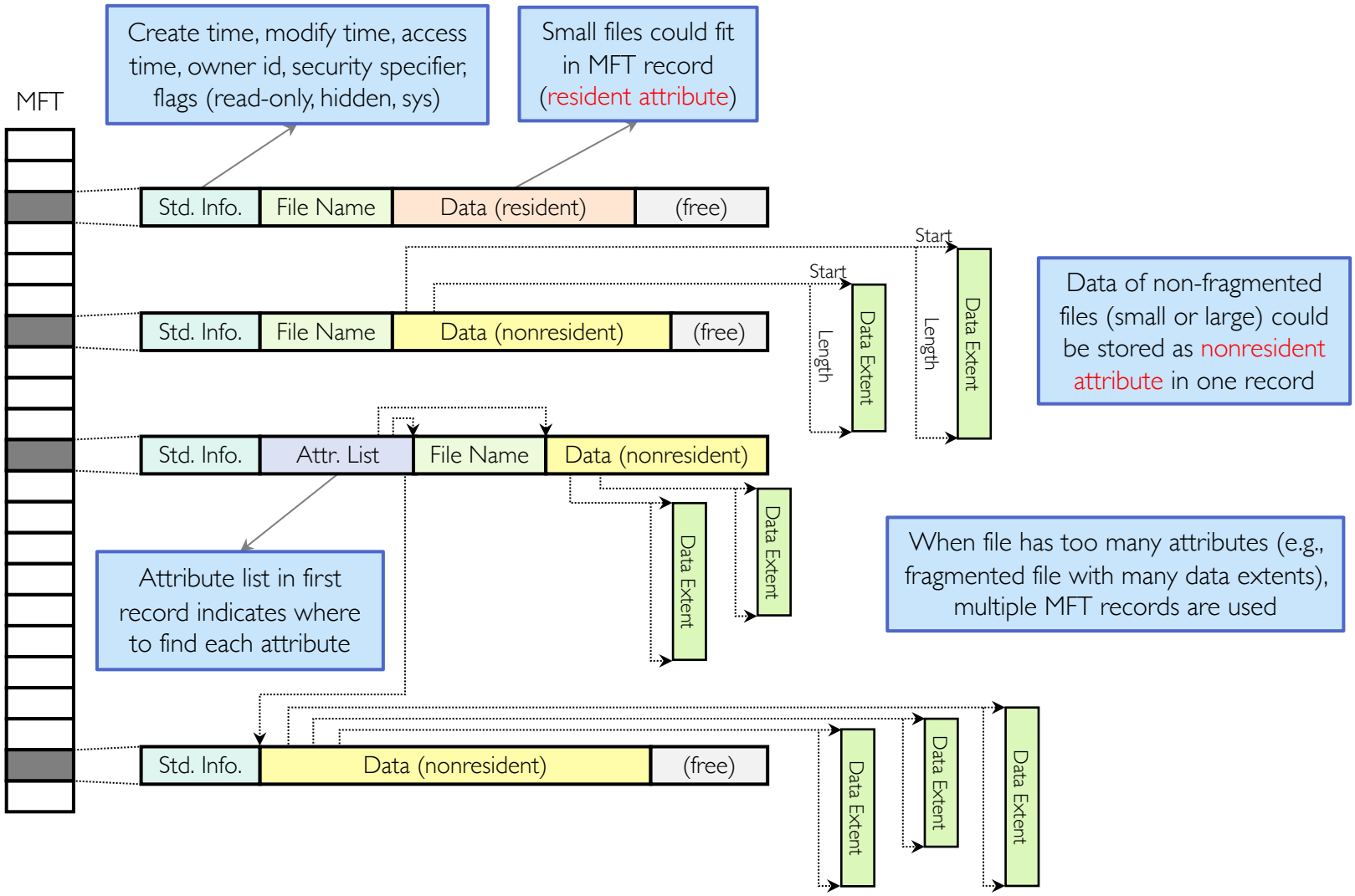
- `open()` performs name resolution
  - Resolves file name, finds inode
  - Creates file descriptor in PCB
  - Returns file handle (another integer) to user process

- `read()`, `write()`, `seek()`, and `sync()` operate on file handle to locate inode
  - Perform appropriate read, write, etc. operations

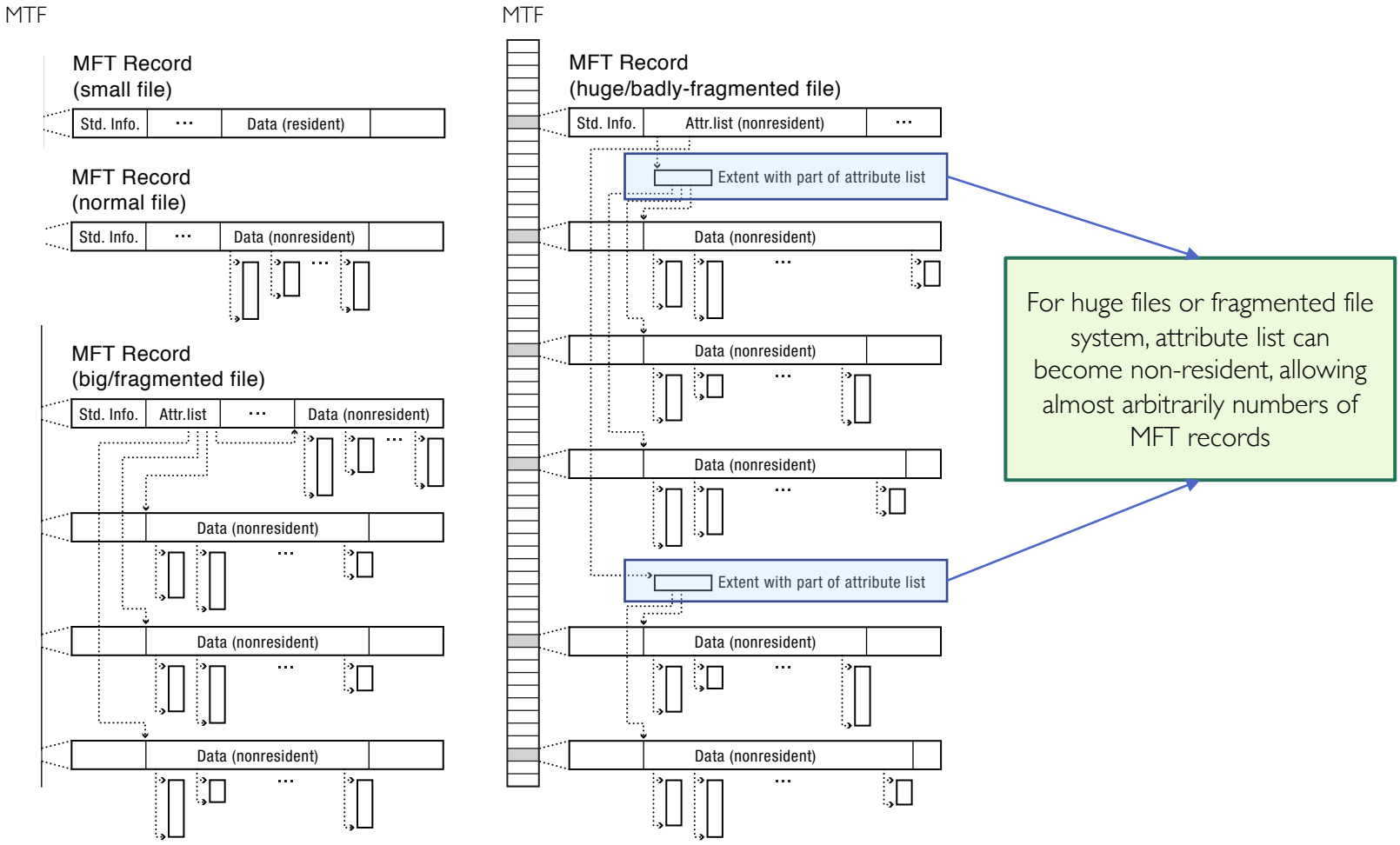
New Technology File System (NTFS)

- Microsoft Windows systems use NTFS by default
- NTFS provides improvements over FAT
  - File metadata, security, and reliability (journaling)
- Extents represent variable-size region of file stored in contiguous region of storage device
  - Similar approach in Linux (ext4)
- Files are represented by variable-depth trees containing pointers to file's extents
  - File with small # of extents is stored in shallow tree even if file is large
  - Deeper trees are only needed for fragmented files
- Roots of trees are stored in Master File Table (MFT)
  - Like inode array
- Directories are organized in B+tree structure

Master File Table (MFT)



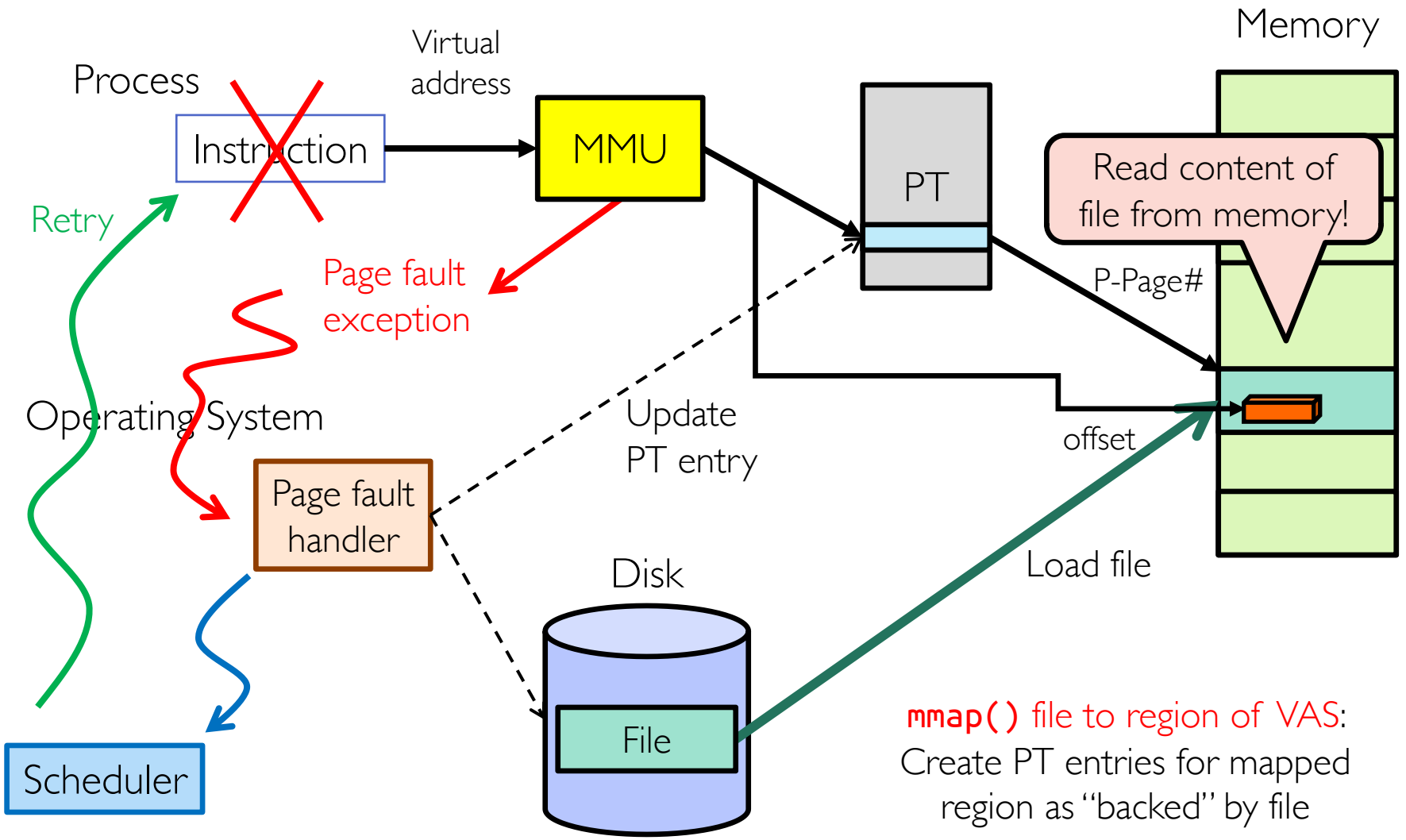
MTF Record Stages



Memory Mapped Files

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

Next Up: What Happens When …



`mmap()` file to region of VAS:
Create PT entries for mapped region as "backed" by file

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

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 unsigned int) &something);
  printf("Heap at: %16lx\n", (long unsigned int) malloc(1));
  printf("Stack at: %16lx\n", (long unsigned int) &mfile);

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

  /* map the file */
  mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, myfd, 0);
  if (mfile == MAP_FAILED) {perror("mmap failed"); exit(1);}

  printf("mmap at: %16lx\n", (long unsigned int) mfile);

  puts(mfile);
  strcpy(mfile+20,"Let's write over it");
  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
```

File System Design Options

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, assym)	Tree (dynamic)
Granularity	Block	Block	Extent
Free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	Defragmentation	Block groups + Reserve space	Extents Best fit defragmentation

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
  - Implemented entirely in OS software (unlike memory caches and TLB)

File System Caching

- Replacement policy?
  - LRU, because we can afford overhead of timestamps for each disk block
  - Advantages
    - Works very well for name translation
    - Works well if memory is big enough to accommodate working set of file blocks
  - Challenges
    - Some apps scan through file system, flushing cache with data used only once
    - E.g., `find . -exec grep foo {} \;`
- Other replacement policies?
  - Some systems allow applications to request other policies
  - E.g., 'Use Once': File system can discard blocks as soon as they are used
- How much memory should OS allocate to buffer cache vs virtual memory?
  - Too much memory to buffer cache ⇒ can't run many apps at once
  - Too little memory to buffer cache ⇒ apps may run slowly (inefficient disk caching)
  - Solution: adjust boundary dynamically so that disk access rates for paging and file access are balanced

File System Caching (cont.)

- Read ahead prefetching: fetch sequential blocks early
  - Fast to access; File system tries to obtain sequential layout
  - Applications tend to do sequential reads and writes
- How much to prefetch?
  - Too many delays on requests by other applications
  - Too few causes many seeks (and rotational delays) among concurrent file requests

Delayed Writes

- Writes not immediately sent to disk
  - So buffer cache is a write back cache
- `write()` copies data from user space to kernel buffer
  - Other apps read data from cache instead of disk
  - Cache is transparent to user programs
- Flushed to disk periodically
  - In Linux: kernel threads flush buffer cache very 30 sec. in default setup
- Some files never actually make it all the way to disk
  - Many short-lived files
- But what if system crashes before buffer cache block is flushed to disk?
- And what if this was for a directory file?
  - Lose pointer to inode
- File systems need recovery mechanisms

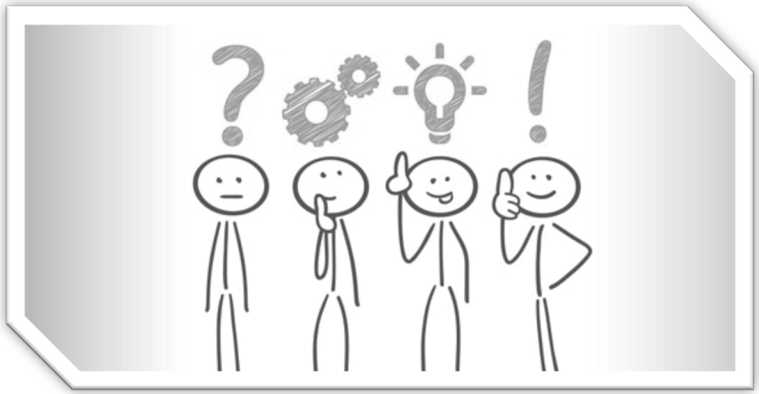
Recall: Important "ilities"

- Reliability: Ability of system or component to perform its required functions under stated conditions for specified time
  - System is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Data must survive system crashes, disk crashes, other problems
- Availability: Probability that system can accept and process requests
  - Can build highly-available systems, despite unreliable components
    - Involves independence of failures and redundancy
  - Often as "nines" of probability. 99.9% is "3-nines of availability"
- Durability: Ability of 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

Summary

- File system
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime
- File Allocation Table (FAT)
  - Linked-list approach
  - Very widely used: Cameras, USB drives, SD cards
  - Simple to implement, but poor performance and no security
- Unix Fast File System (FFS)
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
- New Technology File System (NTFS)
  - Variable extents not fixed blocks, tiny files data is in header

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny