Operating Systems
Lecture 14

fs design

Prof. Mengwei Xu

I/O & Storage Layers

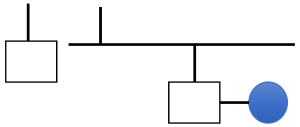
Operations, Entities and Interface

Application / Service

High Level I/O	*streams*
Low Level I/O	*handles*
Syscall	*registers*
File System	*descriptors*
I/O Driver	*Commands and Data Transfers*

`file_open, file_read, … on struct file * & void *`

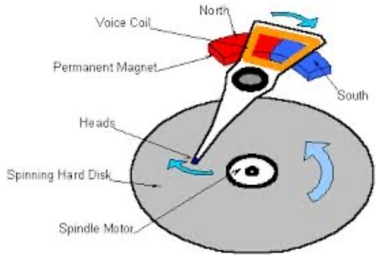
we are here …



Disks, Flash, Controllers, DMA







Layered abstractions of I/O and storage

- Application
  - stdio: fopen(), fclose(), fread(), fwrite()
- Library
  - syscall: open(), close(), read(), write()
- File System
  - *How files and directories are organized in memory and disk*
  - Data block ops between storage and memory
- Block Cache
  - *Caching blocks in memory; write buffering, synchronization*
  - Block device interface: a standard interface for different I/O devices to R/W in fixed-sized blocks (e.g., 512 bytes).
- Device Driver
  - *Translate I/O abstractionsinto device-specific I/O operations*
- Memory-Mapped I/O, DMA, Interrupts
  - *Memory-mapped I/O: maps each device's control registers to a range of physical addresses on the memory bus. For example, the OS knows last key pressed by keyboard in a physical address.*
  - *Direct Memory Access: copy a block of data between storage and memory.*
  - *Interrupts are needed so OS knows when I/O device completes its request (otherwise use polling).*
- Physical Devices

Recall: C Low level I/O

- File Descriptors – as OS object representing the state of a file
  - User has a "handle" on the descriptor

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int create (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html

Recall: C Low level I/O

- File Descriptors – as OS object representing the state of a file
  - User has a "handle" on the descriptor

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written
off_t lseek (int filedes, off_t offset, int whence)
 - set the file offset
   * if whence == SEEK_SET: set file offset to "offset"
   * if whence == SEEK_CRT: set file offset to crt location + "offset"
   * if whence == SEEK_END: set file offset to file size + "offset"
int fsync (int fildes)
 – wait for i/o of filedes to finish and commit to disk
void sync (void) – wait for ALL to finish and commit to disk
```

- When write returns, data is on its way to disk and can be read, but it may not actually be permanent!

Building a File System

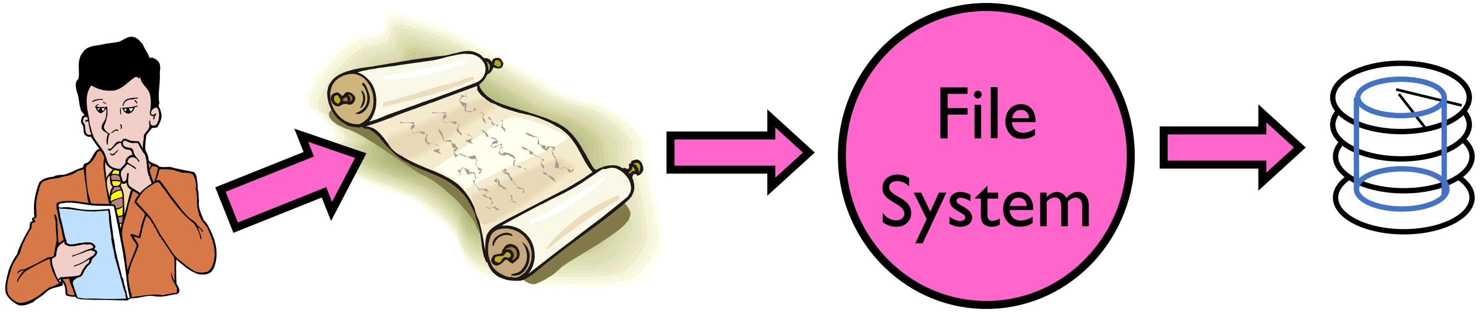
- **File System**: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

- File System Components
  - **Naming**: Interface to find files by name, not by blocks
  - **Disk Management**: collecting disk blocks into files
  - **Protection**: Layers to keep data secure
  - **Reliability/Durability**: Keeping of files durable despite crashes, media failures, attacks, etc.

User vs. System View of a File

- User's view:
  - Durable Data Structures
- System's view (system call interface):
  - Collection of Bytes (UNIX)
  - Doesn't matter to system what kind of data structures you want to store on disk!
- 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

Translating from User to System View



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

Disk Management Policies (1/2)

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

- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector], sort in cylinder-major order
    - ❑ Used in BIOS, but not in OSes anymore
  - Logical Block Addressing (LBA, 逻辑块寻址): Every sector has integer address from zero up to max number of sectors
  - Controller translates from address ⇒ physical position
    - ❑ First case: OS/BIOS must deal with bad sectors
    - ❑ Second case: hardware shields OS from structure of disk

Disk Management Policies (2/2)

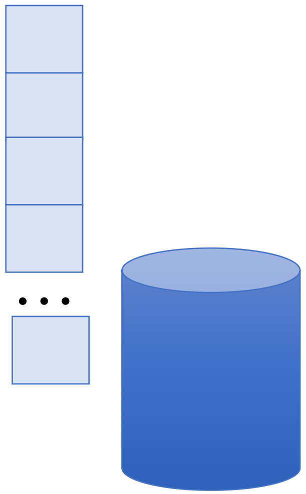
- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk

- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

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

Data blocks



File handle

File descriptor

Fileobject (inode)

Position

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

- Conventions of directory
  - Root directory (根目录): "/"
  - Home directory (主目录): "~/cur_dir/file.txt"
  - Absolute path (绝对路径): "/home/mwx/cur_dir/file.txt"
  - Relative path (相对路径): "file.txt"
- Volume (卷): a collection of physical storage resources that form a logical storage device. Could be a part of or many physical devices.
- Mount (挂载): an operation that creates a mapping from some path in the existing file system to the root directory of the mounted volume's file system

mount –t *type* *device* *dir*

Directory

```
mwx@Dragon21:~$ findmnt -t ext4
TARGET             SOURCE                   FSTYPE    OPTIONS
     /             /dev/sda6                ext4      rw,relatime,errors=remount-ro
├─/data2           /dev/sdc                 ext4      rw,relatime
├─/data            /dev/sdb1                ext4      rw,relatime
├─/var/lib/snapd   /dev/sdc[/zl/snap/snapd] ext4      rw,relatime
└─/boot            /dev/sda1                ext4      rw,relatime
```

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

Components of a file system

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

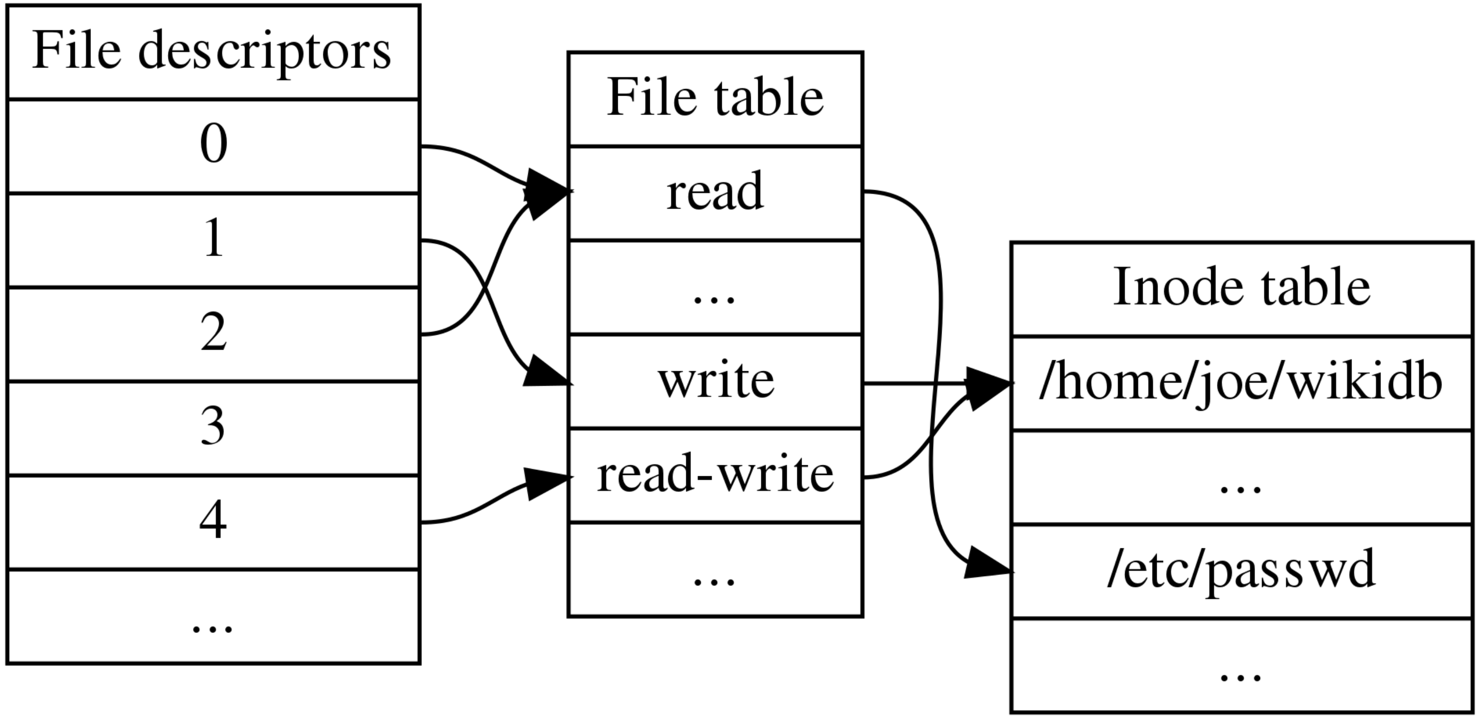
offset — directory — *offset* — index structure

- 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

inode

- An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data.
  - File type
  - Permissions
  - Owner ID
  - Group ID
  - Size of file
  - Time last accessed
  - Time last modified
  - Soft/Hard Links
  - Access Control List (ACLs)



inode

- An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data.
  - File type
  - Permissions
  - Owner ID
  - Group ID
  - Size of file
  - Time last accessed
  - Time last modified
  - Soft/Hard Links
  - Access Control List (ACLs)

```
❯ ls -i
2698698 awards.html                  2698800 materials
2698699 css-js                       2698803 old-materials
2698708 files                        3988068 papers.html
2698786 full-conference.json         2698809 projects
2698787 full-journal.json            2698817 selected-conference.json
2698788 full-pub.html                2698818 selected-journal.json
2698789 image                        2698819 service.html
2698797 index-shenzhen.html          2698820 students.html
2698798 index.html                   2698821 teaching.html
2698799 invited-talks.html
```

inode

- An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data.

- Each file has exactly one corresponding one inode? (i.e., 1-1 mapping)
  - True for most traditional Unix-like filesystems
  - No true with hard links (covered later)

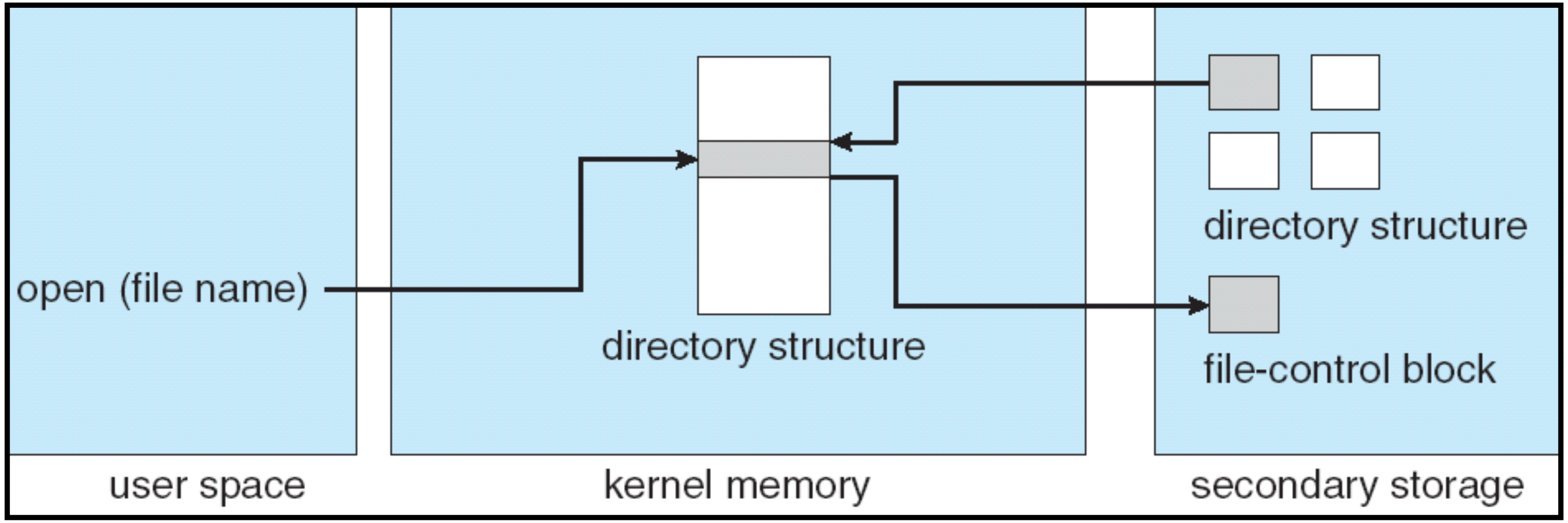
- When a file is copied – a new inode
- When a file is moved – nothing changed
  - Unless to another filesystem

How Many inodes in Linux

- For 32-bit inode number, it's 2^32 (about 4 billions)
  - Max
- It's also configurable in many file systems
- *Out of inode* error..

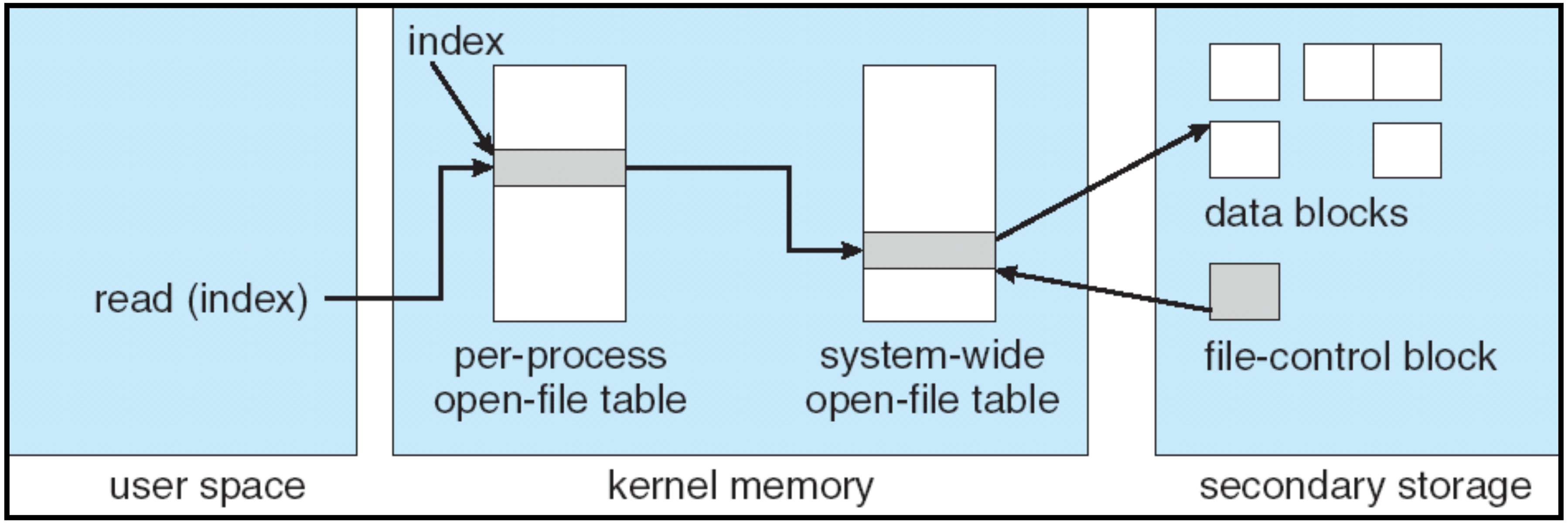
```
echo:homepage echo$ df -i
Filesystem    512-blocks      Used Available Capacity iused      ifree %iused
/dev/disk1s1  1953595632  21968928 991671656     3%  488378 9767489782    0%
devfs                387       387         0   100%     678          0  100%
/dev/disk1s2  1953595632 934163472 991671656    49% 4233888 9763744272    0%
/dev/disk1s5  1953595632   4194344 991671656     1%       2 9767978158    0%
map auto_home          0         0         0   100%       0          0  100%
```

In-Memory File System Structures



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

In-Memory File System Structures



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

Typical File Systems

- FAT (Microsoft File Allocation Table), 1970s.
  - Extremely simple index structure: a linked list.
  - Still widely used in devices like flash memory sticks and digital cameras
- FFS (Unix Fast File System), 1980s.
  - Tree-based multilevel index to improve random access efficiency.
  - Uses a collection of locality heuristics to get good spatial locality.
  - EXT2 and EXT3 are based on FFS.
- NTFS (Microsoft New Technology File System): 1990s.
  - More flexible tree structure.
  - Mainstream file system on MS.
  - It's representative to EXT4, XFS, and Apple's Hierarchical File Systems (HFS and HFS+).

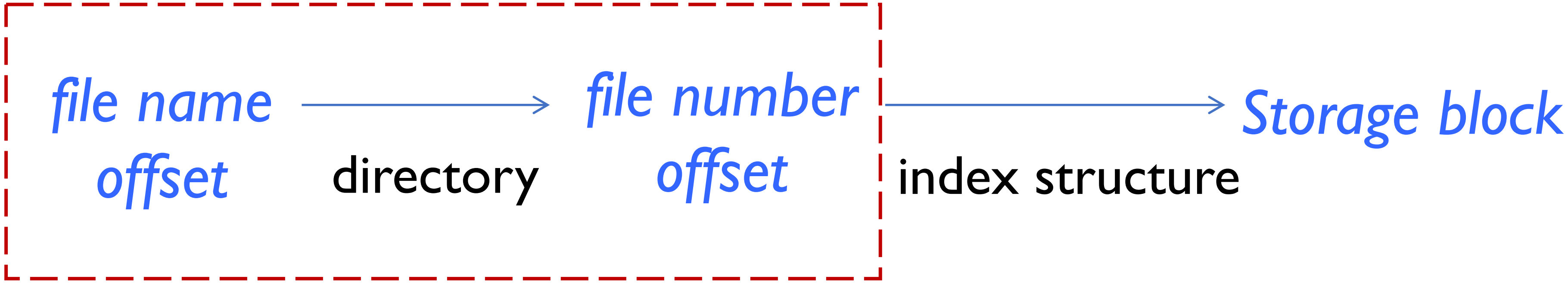
Goals for Today

- Directories: naming data
  - How do we convert a file name to the file number?
- Files: finding data
  - How do we locate storage block based on file number?
- Virtual file systems (VFS)
  - How do we make different FSs work together easily?

file name offset —directory→ *file number offset* —index structure→ *Storage block*

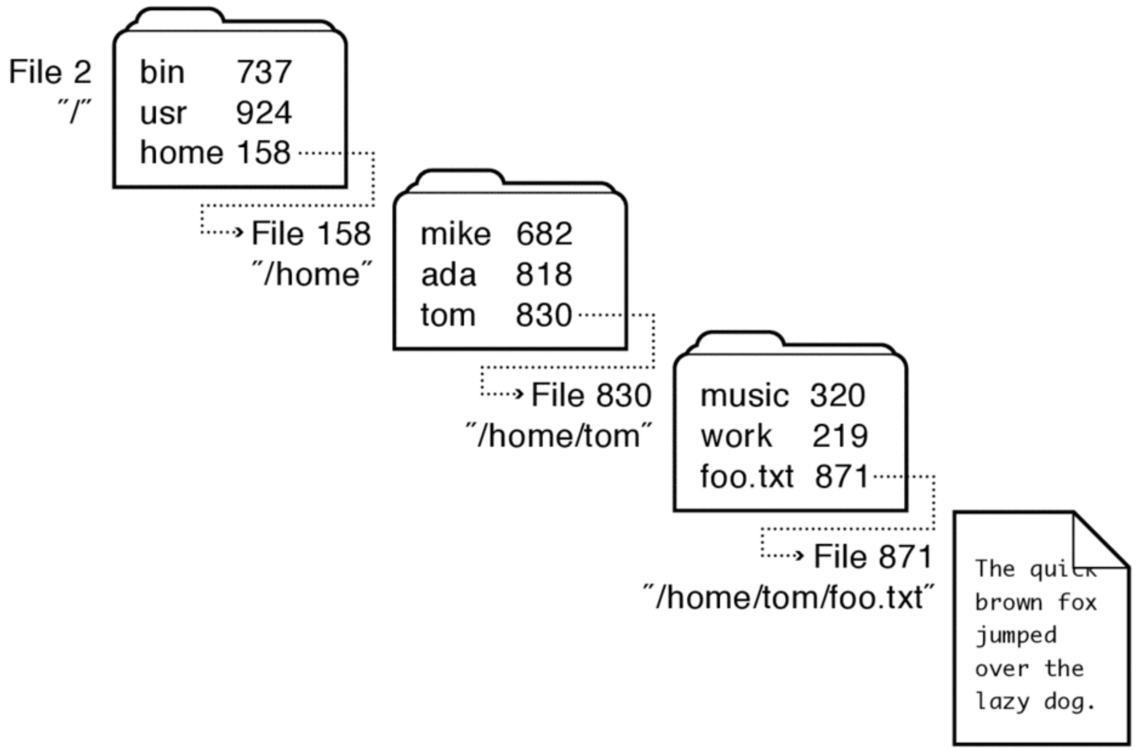
Goals for Today

- Directories: naming data
  - How do we convert a file name to the file number?
- Files: finding data
  - How do we locate storage block based on file number?
- Virtual file systems (VFS)
  - How do we make different FSs work together easily?



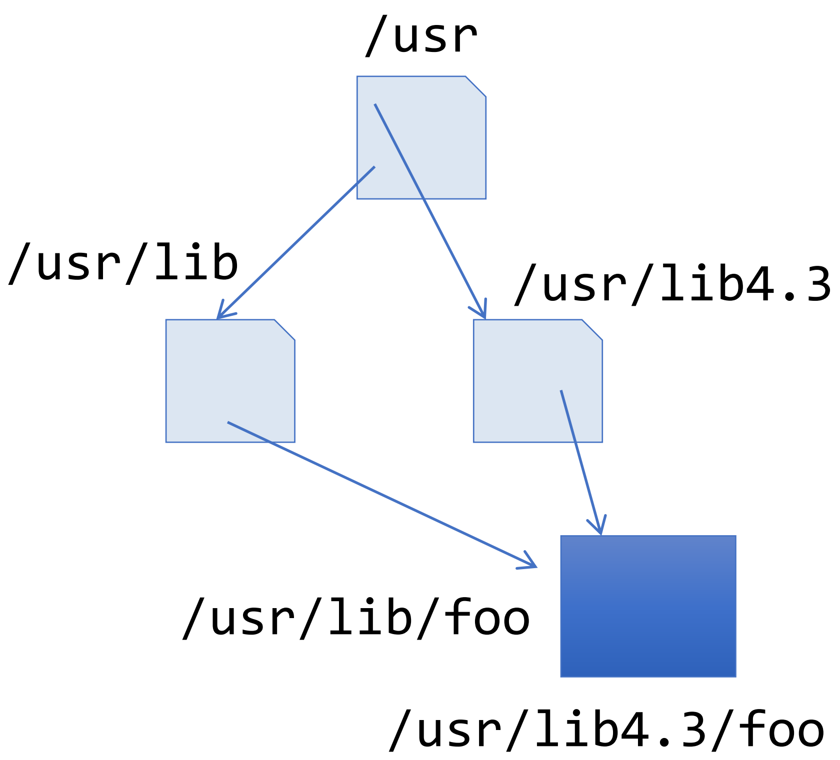
Directory Structure

- Directory is treated as a file with a list of <file name: file number> mappings
- The file number of the root directory is agreed ahead of time
  - In many Unix FSs, it's 2.



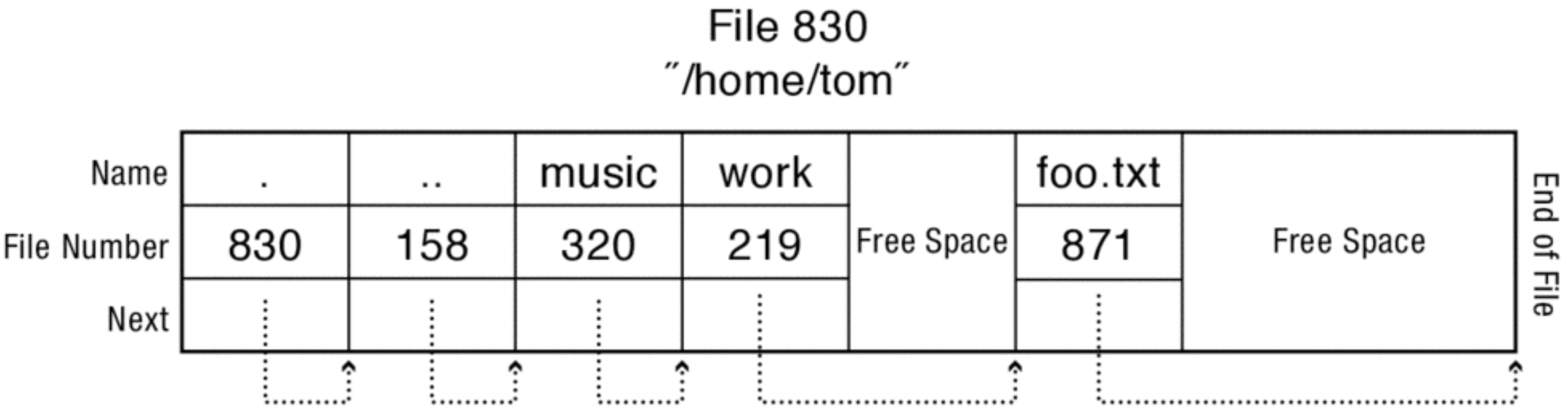
Directory Operations

- Stored in files, can be read, but typically don't
  - System calls to access directories
  - `open` / `creat` traverse the structure
  - `mkdir` /`rmdir` add/remove entries
  - `link` / unlink (`rm`)
    - ❑ Link existing file to a directory
      - Not in FAT !
    - ❑ Forms a DAG
- When can file be deleted?
  - Maintain ref-count of links to the file
  - Delete after the 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)`



Directory Internals

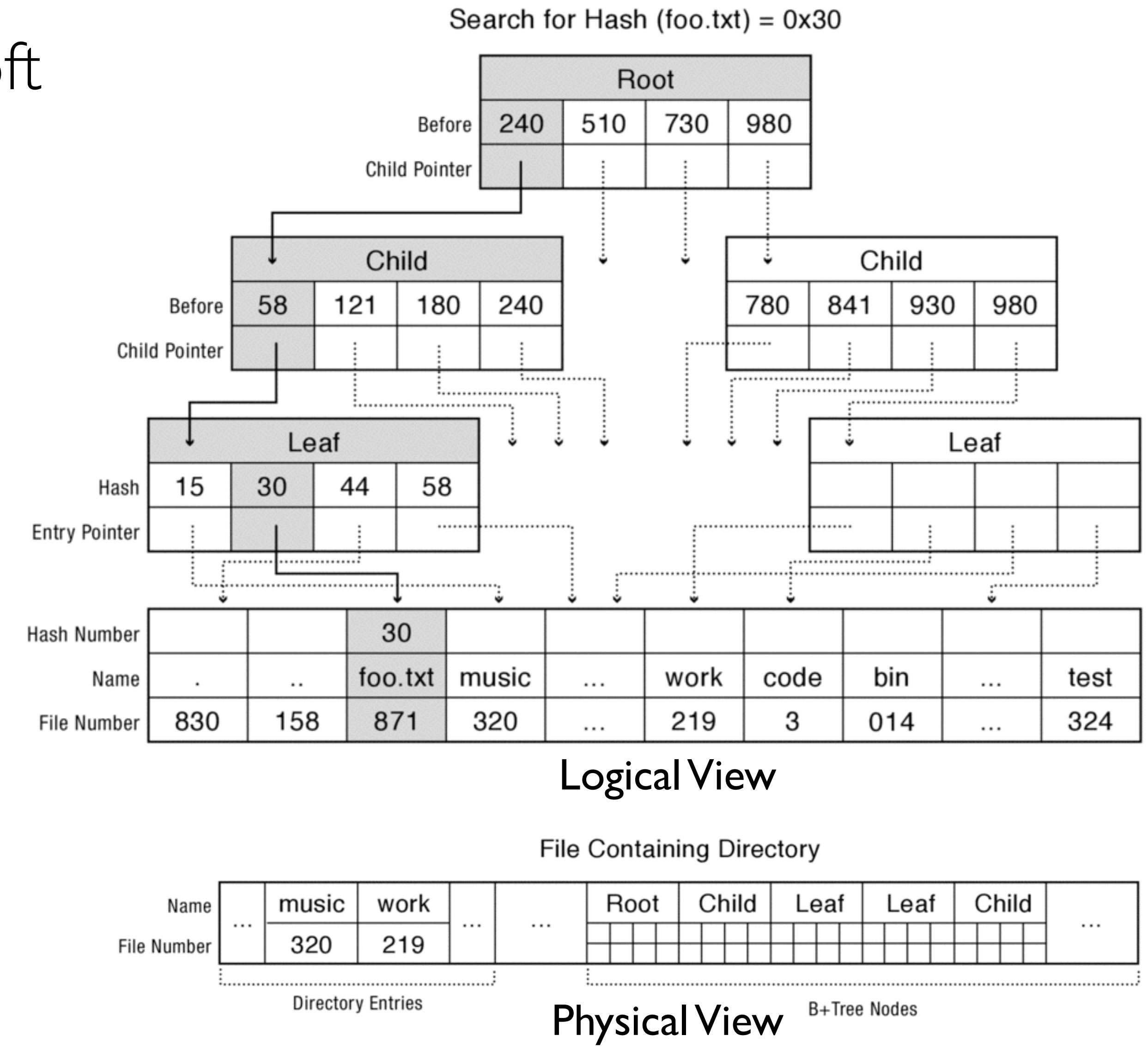
- Early implementations simply stored linear lists of <file name, file number> in directory files.
  - Free spaces are for new entries. Note: files can be added/deleted.



- Works fine in most cases. But when there are thousands of files in a directory.. The access could be slow!

Directory Internals

- Modern FSs (Linux XFS, Microsoft NTFS, and Oracle ZFS) organize directory's contents as a tree.
  - B/B+ tree: fast lookup, insert, and removal
  - Names are first hashed into a key, which is used to find the file number in the tree



Directory Structure Access Cost

- 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 (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

Hard Link

- *ln* command – link()
  - It creates another name in the directory you are creating the link to, and refers it to the same inode number of the original file.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
prompt> ls -i file file2
67158084 file
67158084 file2
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

Hard Link

- *ln* command – link()
  - It creates another name in the directory you are creating the link to, and refers it to the same inode number of the original file.
  - OS maintains a reference count for each inode.

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084        Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084        Links: 2 ...
prompt> stat file2
... Inode: 67158084        Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084        Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084        Links: 2 ...
```

Soft Link

- *ln -s* command – soft (or symbolic) link()
  - A special type of file (as against regular file/dir) whose contents are the pathname of the linked-to file.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello

prompt> ls -al
drwxr-x---    2    remzi remzi 29      May 3 19:10    ./
drwxr-x---    27   remzi remzi 4096    May 3 15:14    ../
-rw-r-----    1    remzi remzi 6       May 3 19:10    file
lrwxrwxrwx    1    remzi remzi 4       May 3 19:10    file2 -> file
```

Soft Link

- *ln -s* command – soft (or symbolic) link()
  - A special type of file (as against regular file/dir) whose contents are the he pathname of the linked-to file.

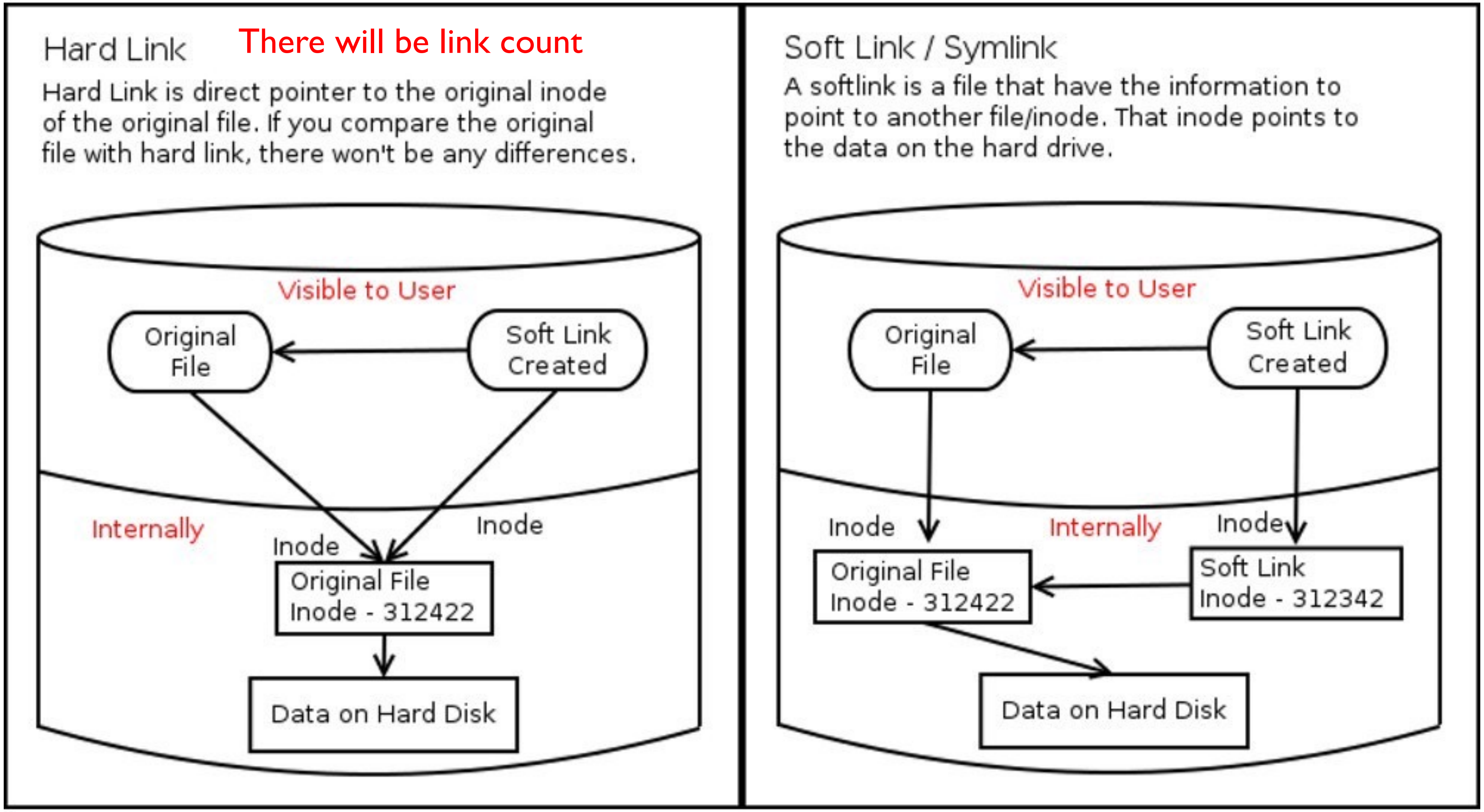
```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r-----      1 remzi remzi 6      May 3 19:17 alongerfilename
lrwxrwxrwx      1 remzi remzi 15     May 3 19:17 file3 -> alongerfilename
```

Soft Link

- *ln -s* command – soft (or symbolic) link()
  - A special type of file (as against regular file/dir) whose contents are the he pathname of the linked-to file.
  - Dangling reference

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Hard Link vs. Soft Link (Symlink)

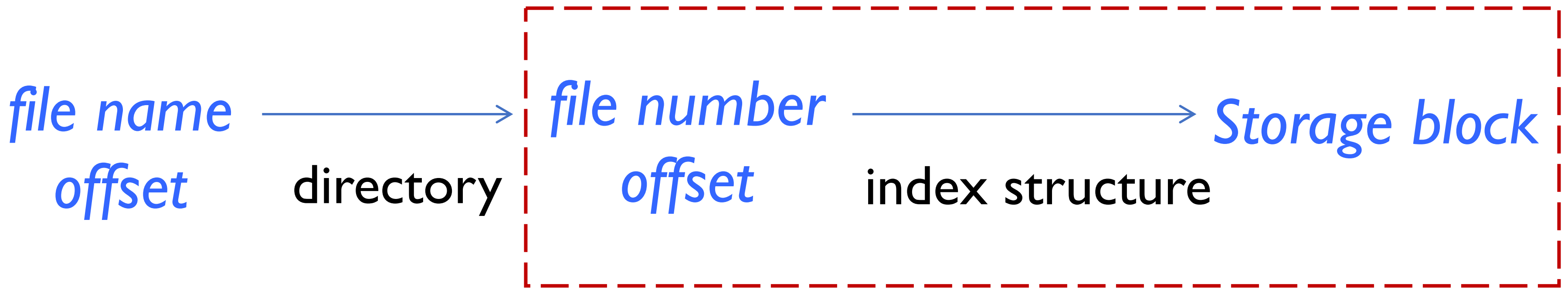


Hard Link vs. Soft Link (Symlink)

- When shall I use hard link
  - I don't want to increase inode number.
  - I want the linked file/directory keep working even when the original file/directory is deleted.
  - Version control.
- When shall I use soft link
  - I want to link to a directory.
    - ❑Using hard link to directory might result in cycle in the directory tree.
  - I want to link to files in other disk partitions.
    - ❑Because inode numbers are only unique within a particular file system, not across file systems.
  - Shortcuts.

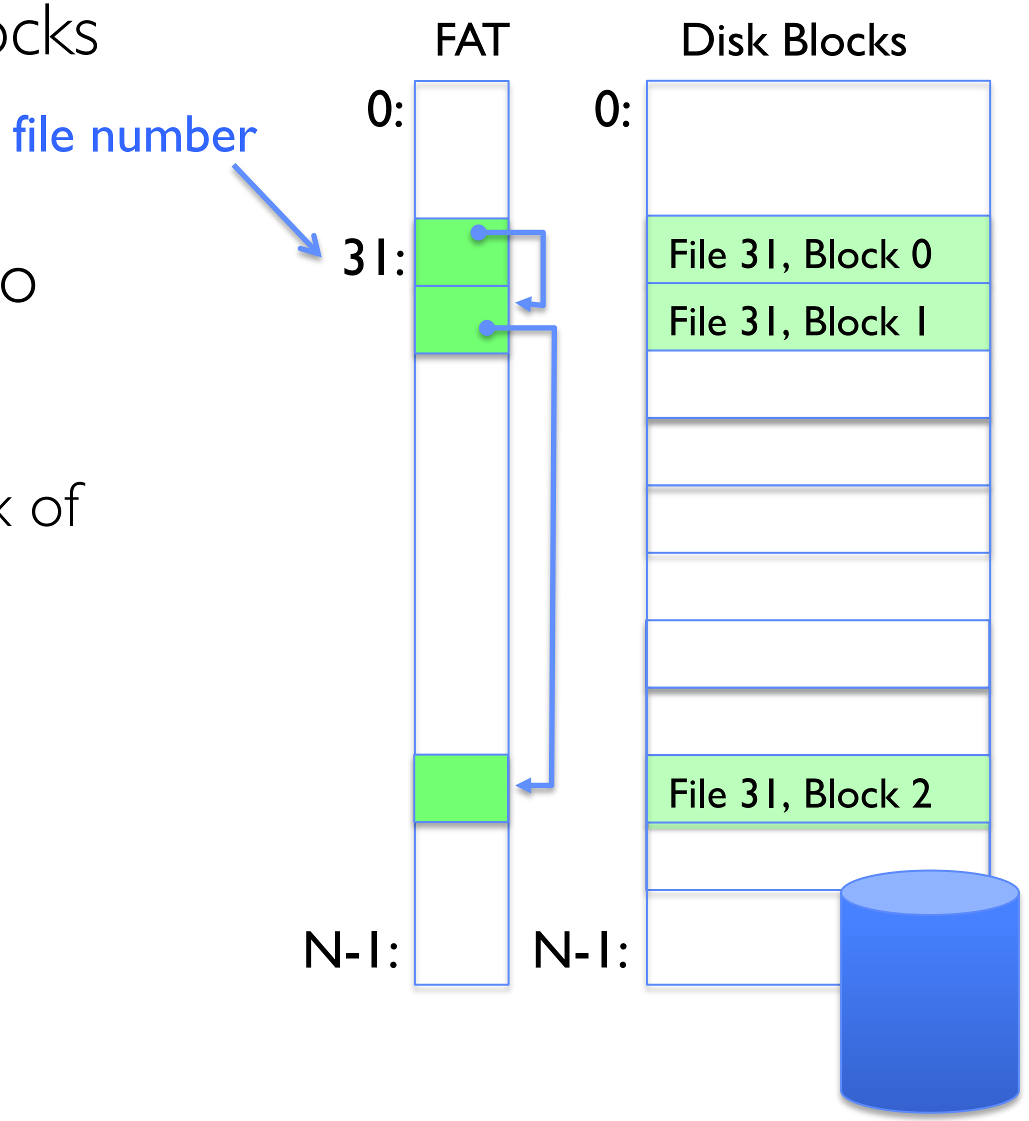
Goals for Today

- Directories: naming data
  - How do we convert a file name to the file number?
- Files: finding data
  - How do we locate storage block based on file number?
- Virtual file systems (VFS)
  - How do we make different FSs work together easily?



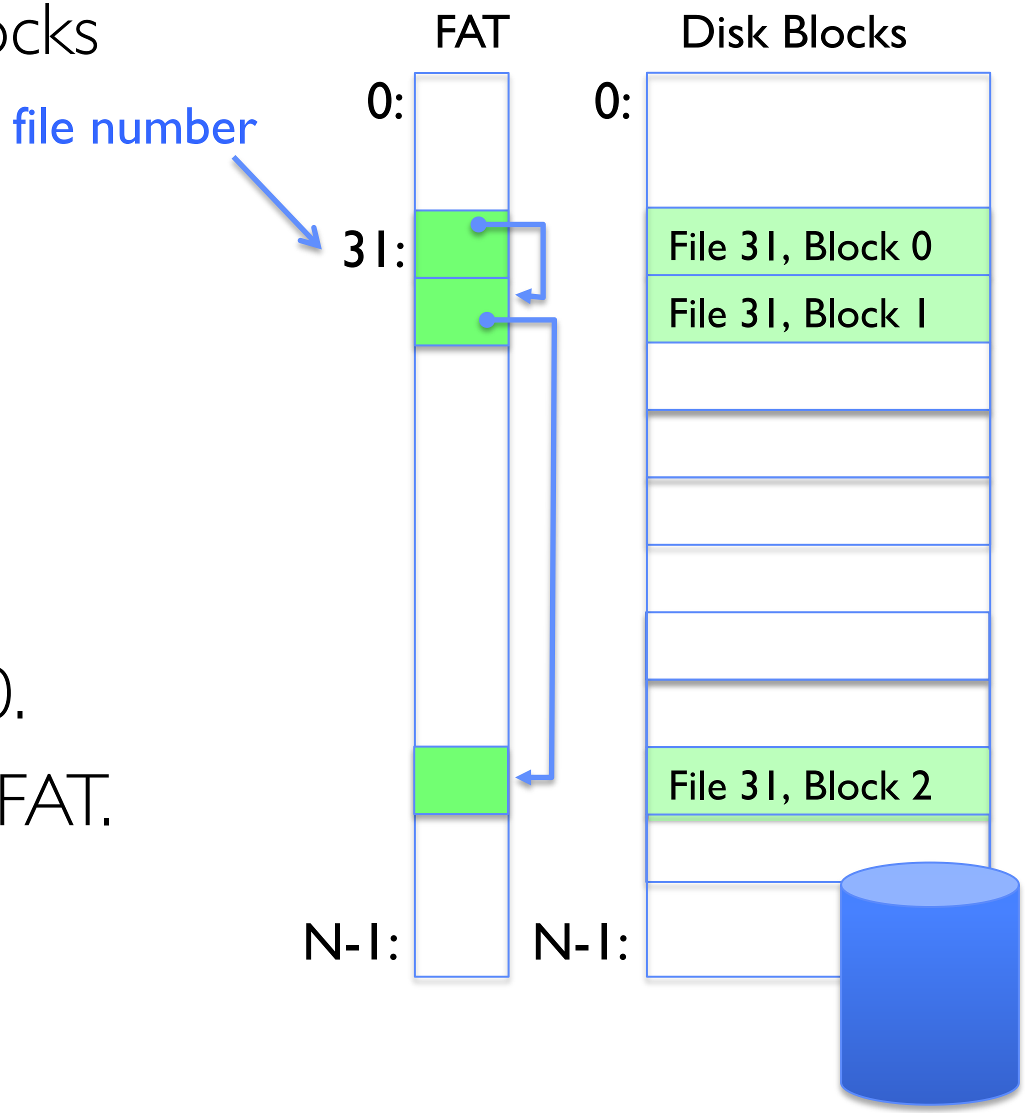
FAT (File Allocation Table)

- FAT is a linked list as 1-1 map with blocks
  - Represented as a list of 32-bit entries
  - Older versions use fewer bits
- Each entry in FAT contains a pointer to the next FAT entry of the same file
  - Or a special END_OF_FILE value.
  - The file number is the 1st (or root) index of the block list for the file
- For File No. #i, its
  - 1st data block index: i
  - 2nd data block index: *(FAT[i])
  - 3rd data block index: *(*(FAT[i]))
  - ..



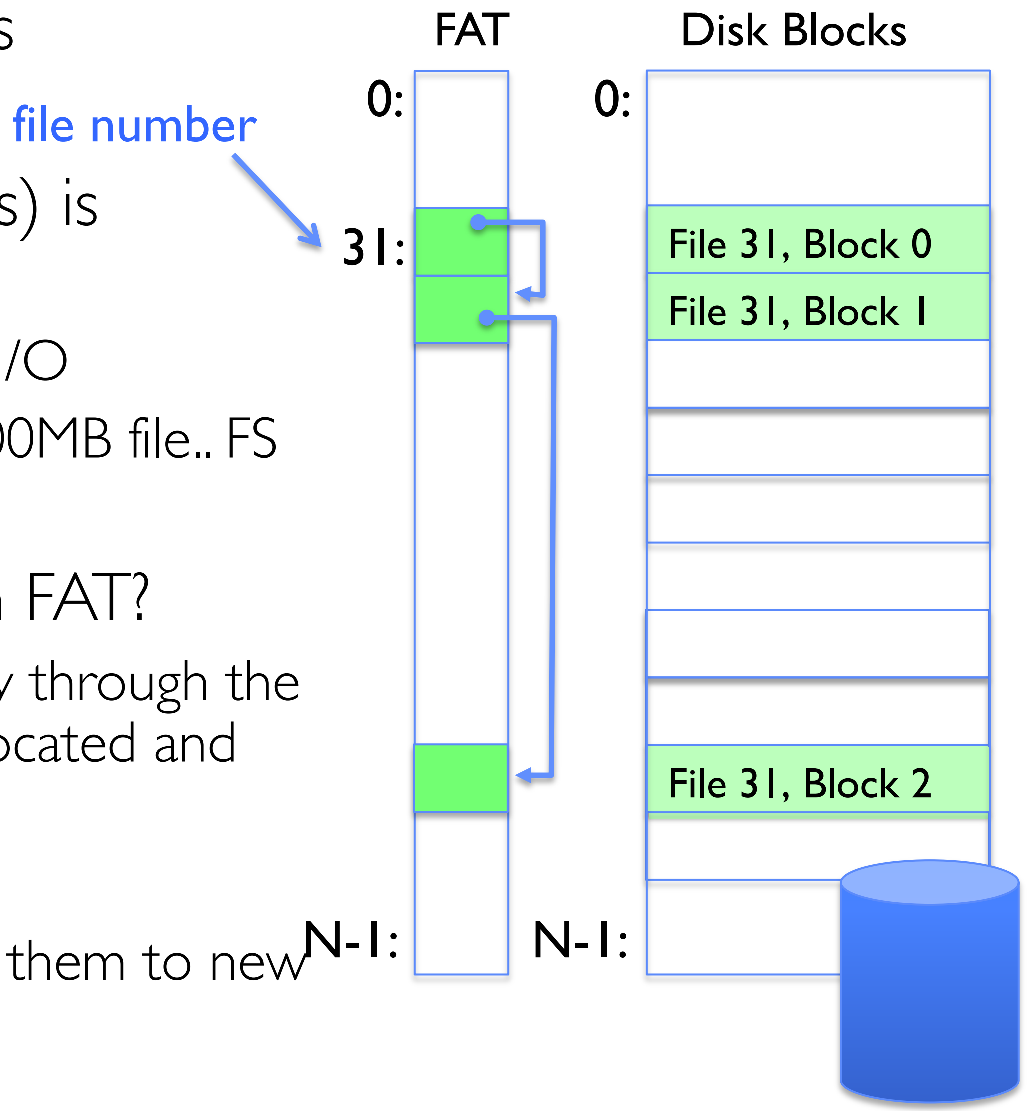
FAT (File Allocation Table)

- FAT is a linked list as 1-1 map with blocks
  - Represented as a list of 32-bit entries
- Where is FAT stored?
  - On Disk, on boot cache in memory, second (backup) copy on disk
- Free space: FAT free list, i.e., FAT[i] = 0.
- To find a free block: scanning through FAT.



FAT (File Allocation Table)

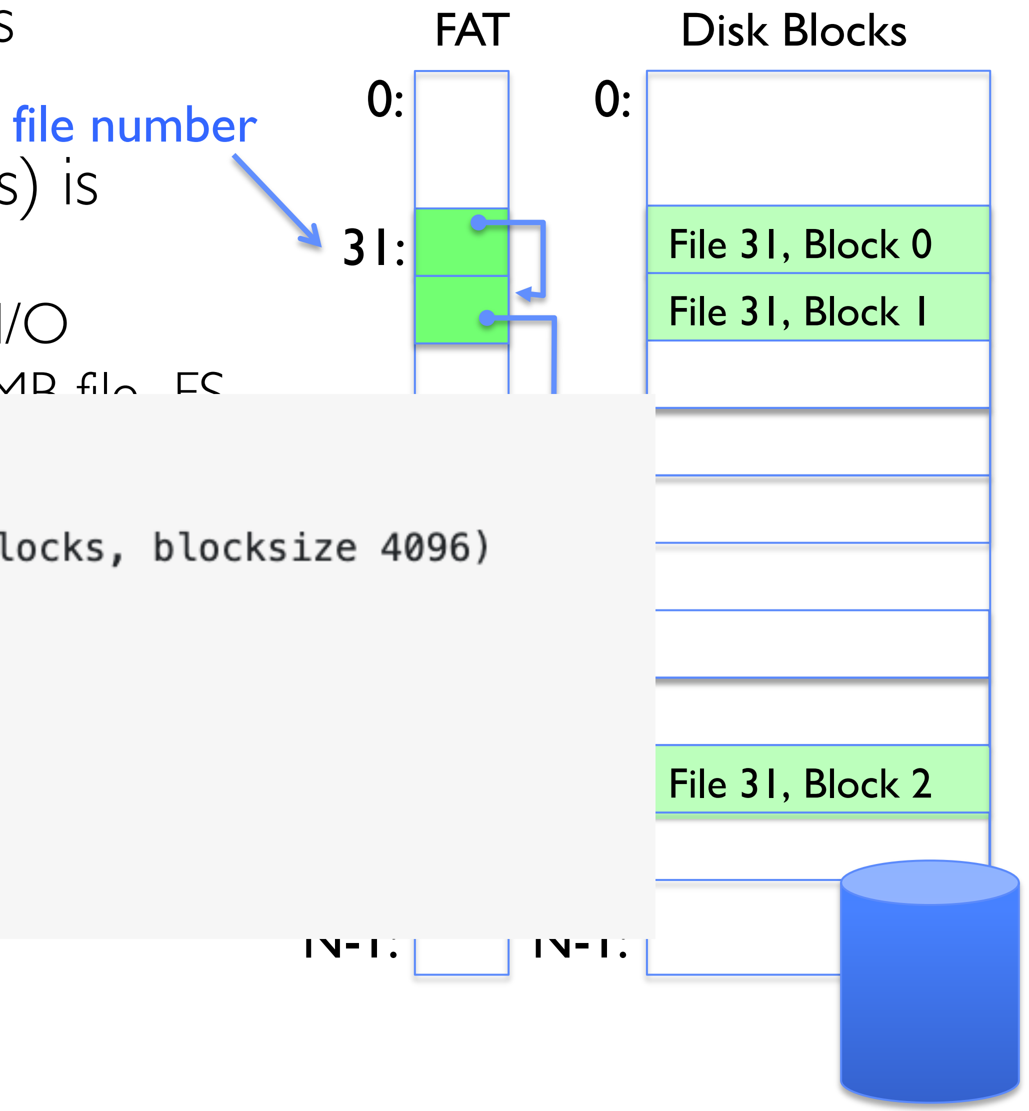
- FAT is a linked list as 1-1 map with blocks
  - Represented as a list of 32-bit entries
- **Locality** (storing a file in sequential blocks) is important for fast I/O
  - Sequential I/O is much faster than random I/O
  - Imagine you want to append 100MB to a 200MB file.. FS cannot guarantee they are stored sequential
- How to ensure good locality heuristics in FAT?
  - Simple strategy: *next fit*, i.e., scans sequentially through the FAT starting from the last entry that was allocated and return the next free entry
  - Still, there will be increasing fragment
  - Defragmentation tool: read files and rewrite them to new locations with better locality



FAT (File Allocation Table)

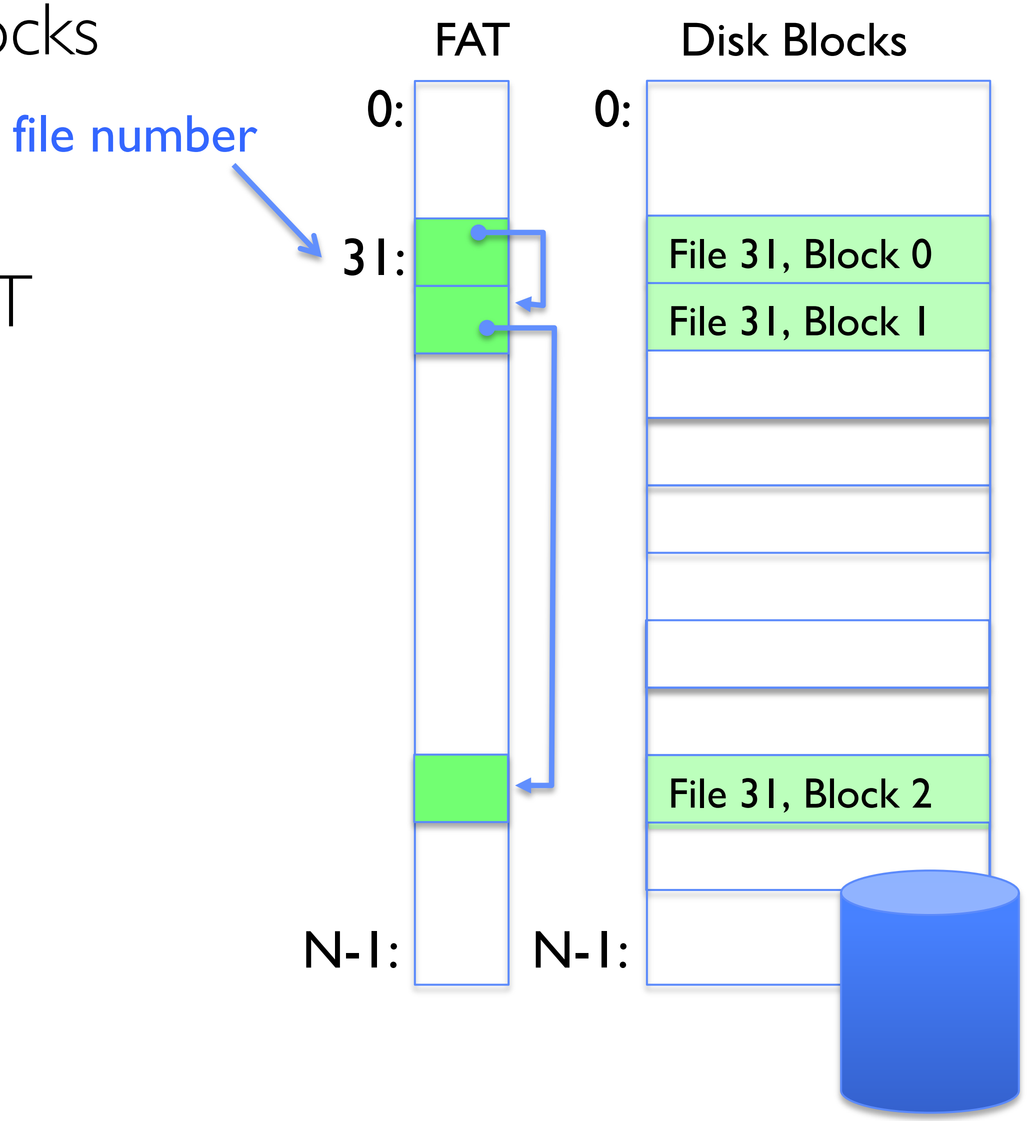
- FAT is a linked list as 1-1 map with blocks
  - Represented as a list of 32-bit entries
- Locality (storing a file in sequential blocks) is important for fast I/O
  - Sequential I/O is much faster than random I/O
  - Imagine you want to write 100MB to a 200MB file. FS
-

```
# filefrag -v /var/log/messages.1
Filesystem type is: ef53
File size of /var/log/messages.1 is 41733 (11 blocks, blocksize 4096)
 ext logical physical expected length flags
   0       0  2130567               1
   1       1 15907576  2130568      1
   2       2 15910400 15907577      1
   3       3 15902720 15910401      7
   4      10  2838546 15902727      1 eof
/var/log/messages.1: 5 extents found
```



FAT (File Allocation Table)

- FAT is a linked list as 1-1 map with blocks
  - Represented as a list of 32-bit entries
- READ: just get block by block with FAT
- WRITE
  - Get blocks from free list
  - Linking them into a file
- Format a disk
  - Zero the blocks, link up the FAT free list
- Quick format
  - Link up the FAT free-list



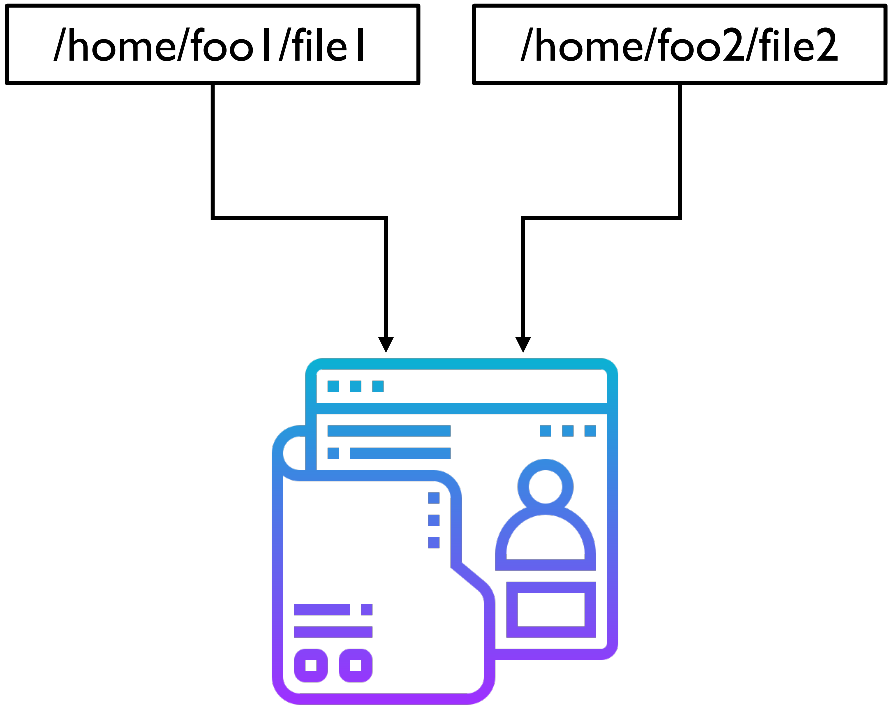
FAT Issues

- Poor locality: there will be fragmentations
- Poor random access: needs to traverse the file's FAT entries till the block is reached
- File metadata stored in directory entries, therefore being limited
  - Only has file's name, size, and creation time, but cannot specify the file's owner or group.
- Limitations on volume and file size
  - With top 4 bits reserved.
  - 2^28 blocks * 4KB block size = 1TB.
  - Larger block size (up to 256KB)?
  - File size is encoded in 32 bits, so less than 4GB.

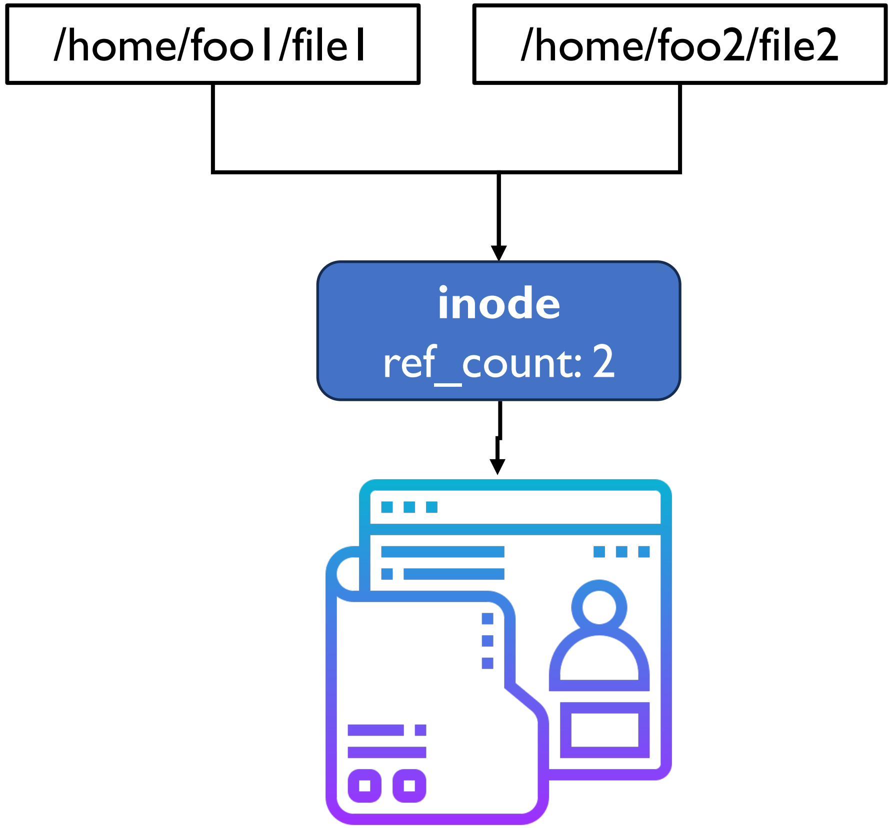
FAT Issues

- No support for hard links: hard to maintain a link count.
  - FAT does not use inode.

FAT



Other Filesystems



Unix File System (1/2)

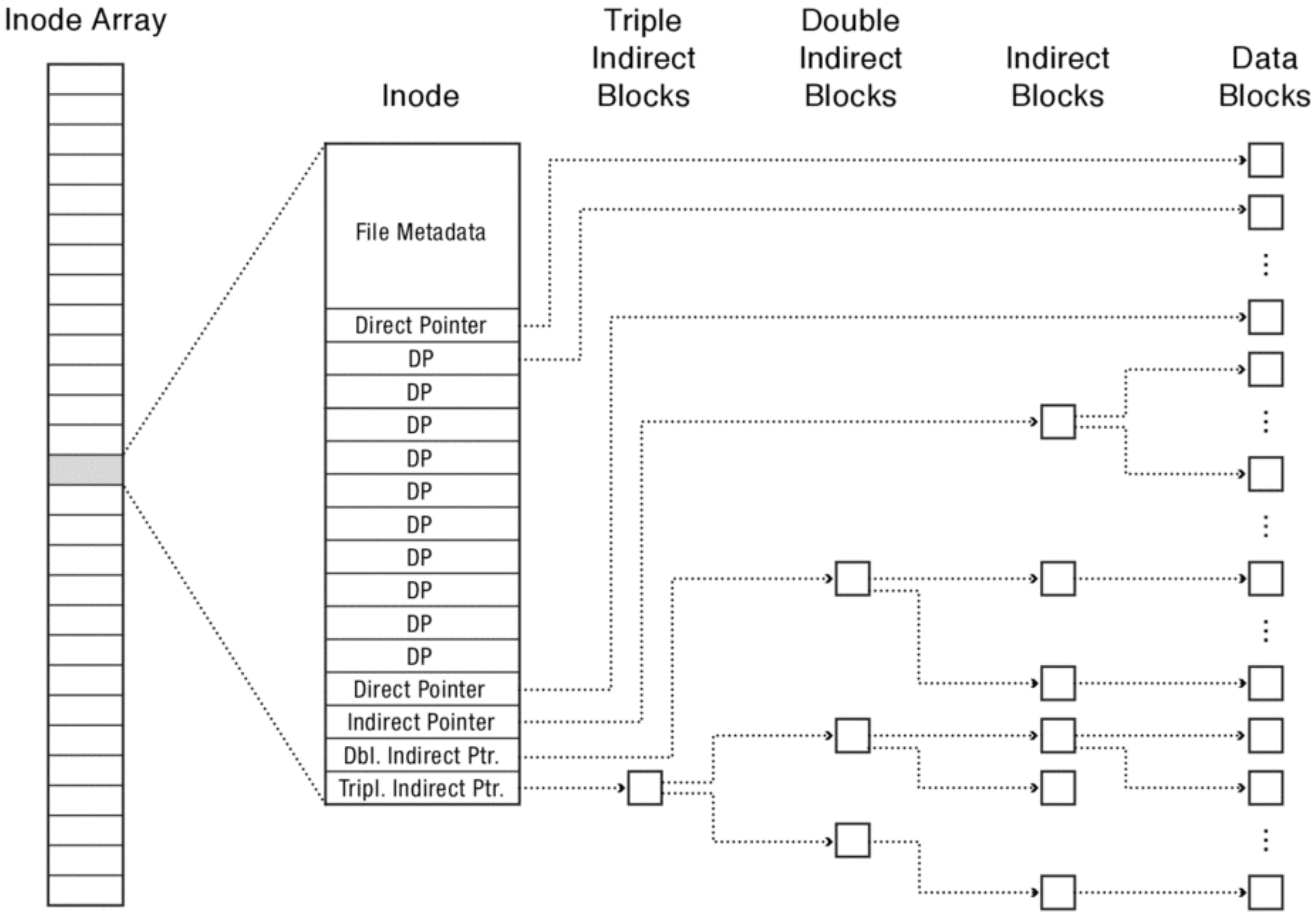
- Original inode format appeared in BSD 4.1
  - Berkeley Standard Distribution Unix
  - Similar structure for Linux Ext2/3
- File Number is index into inode arrays
- Multi-level index structure
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks

Unix File System (2/2)

- Metadata associated with the file
  - Rather than in the directory that points to it
- UNIX Fast File System (FFS) BSD 4.2 Locality Heuristics:
  - Block group placement
  - Reserve space
- Scalable directory structure

File Attributes

- Multi-level index
  - Fixed, asymmetric tree

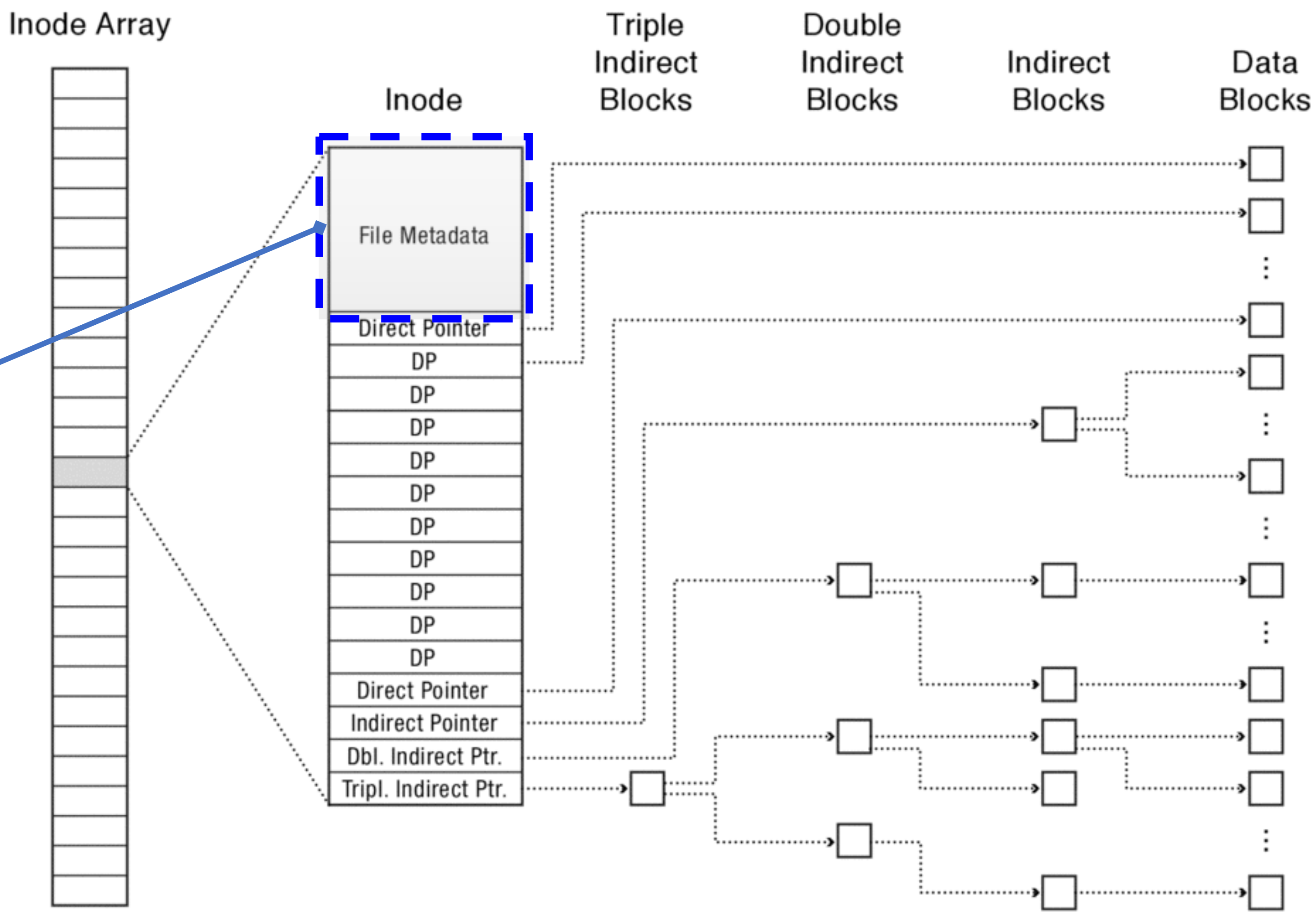


File Attributes

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



File Attributes

Inode Array Triple Double

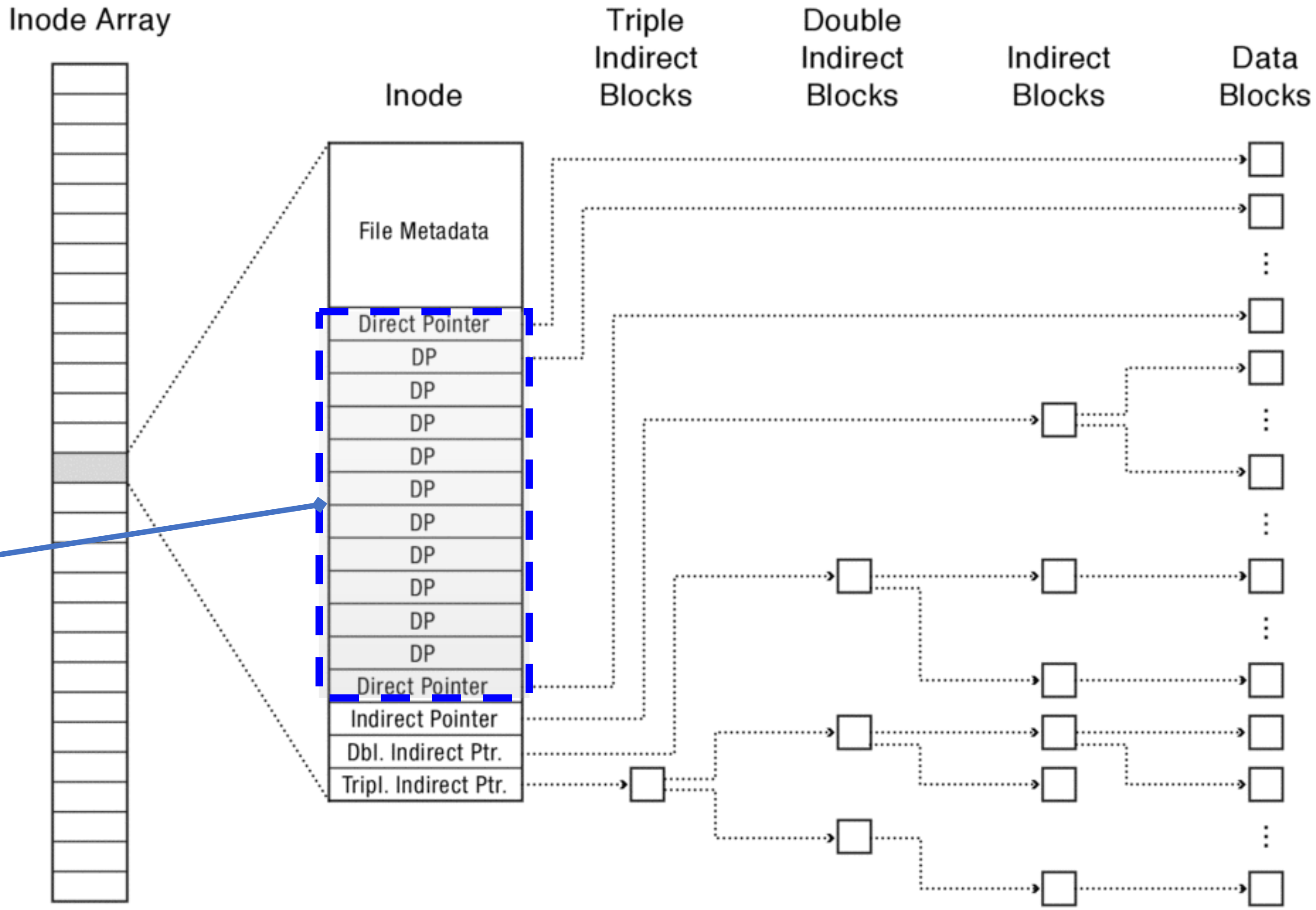
```
echo:homepage echo$ ls -la
total 176
drwxr-xr-x@ 11 echo  staff    352 Nov  2 13:55 .
drwxr-xr-x@ 10 echo  staff    320 Nov  2 13:55 ..
-rw-r--r--@  1 echo  staff   6148 Nov  2 13:45 .DS_Store
drwxr-xr-x@ 12 echo  staff    384 Nov  2 13:55 .git
-rw-r--r--@  1 echo  staff   1374 Jul  5 09:55 awards.html
drwxr-xr-x@ 48 echo  staff   1536 Nov  2 13:47 files
drwxr-xr-x@  8 echo  staff    256 Dec  9  2021 image
-rwxr-xr-x@  1 echo  staff  55677 Nov  2 13:48 index.html
-rw-r--r--@  1 echo  staff  18233 Jun 22  2021 index.old.html
drwxr-xr-x@  4 echo  staff    128 Jan  8  2022 materials
drwxr-xr-x@  6 echo  staff    192 Dec  9  2021 projects
```

User
Group
9 basic
- UG
Setuid
- ex
ra
Setgid bit
- execute at group's permissions

File Attributes

12x Direct Pointers (直接索引)

4kB blocks ⇒
sufficient for files up to 48KB



File Attributes

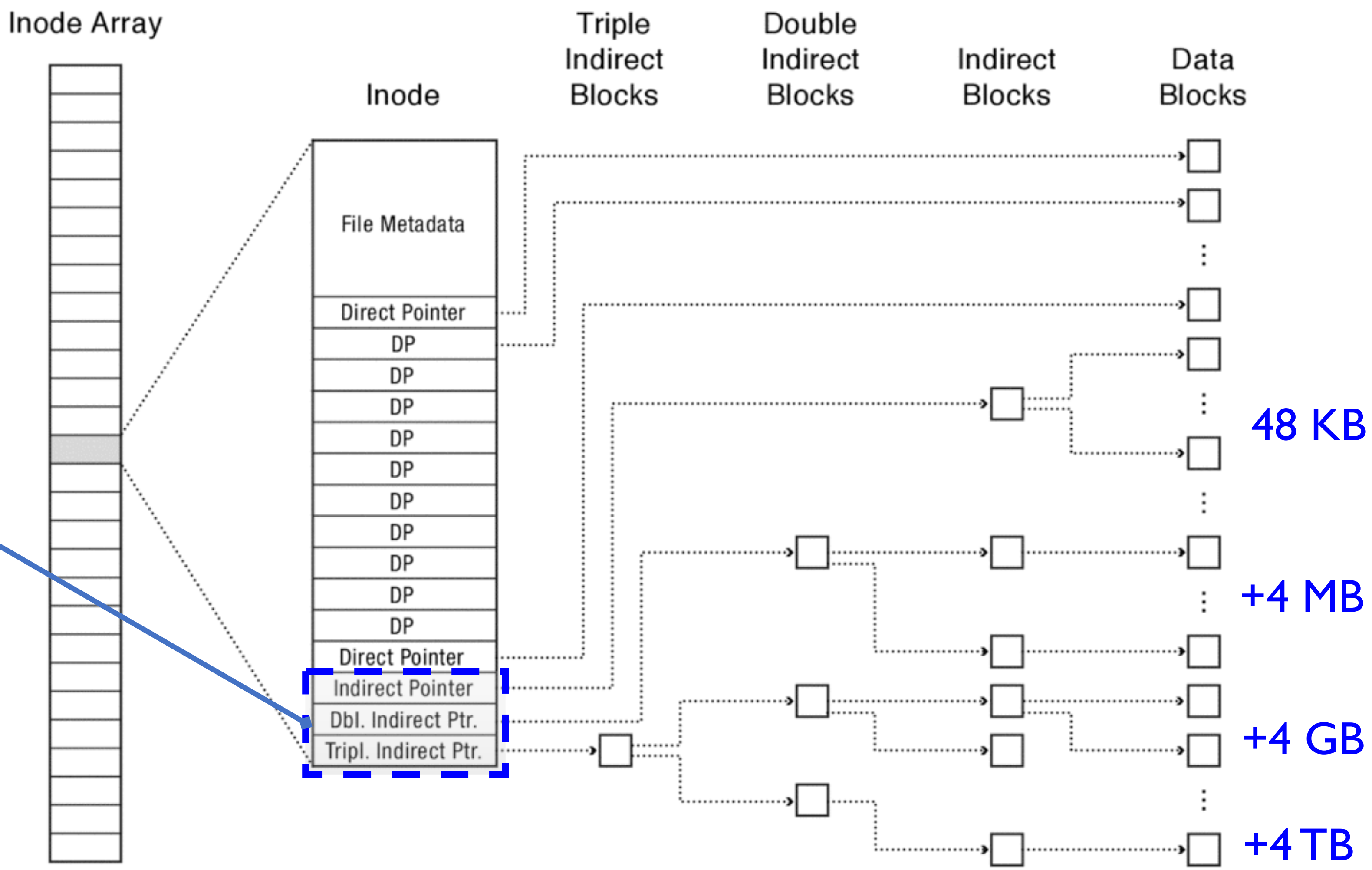
Indirect pointers
- point to a disk block containing only pointers

4 kB blocks => 1024 ptrs

Indirect Pointer (一级间接索引) => 4 MB

Double Indirect Pointer (二级..) => 4 GB

Triple Indirect Pointer (三级..) => 4 TB



FFS Characteristics

- **Tree structure.** Each file is represented as a tree, which allows the file system to efficiently find any block of a file.
- **High degree.** The FFS tree uses internal nodes with many children.
  - A 4KB file block contains 1024x blockID in 4 bytes.
  - Improves sequential reads and writes. Why?
- **Fixed structure.** The FFS tree has a fixed structure.
  - For a given configuration of FFS, the first set of d pointers always point to the first d blocks of a file; etc.
  - Make implementation easier.
- **Asymmetric.** FFS's tree structure is asymmetric, i.e., different depths.
  - Small files can be stored with low cost (size and access speed).
  - While we still support very large files.

Asymmetric vs. Symmetric

- In a symmetric tree with each entry to be triple indirect pointers

$\Rightarrow$To store a 4B file, how much space we need?

Asymmetric vs. Symmetric

- In a symmetric tree with each entry to be triple indirect pointers

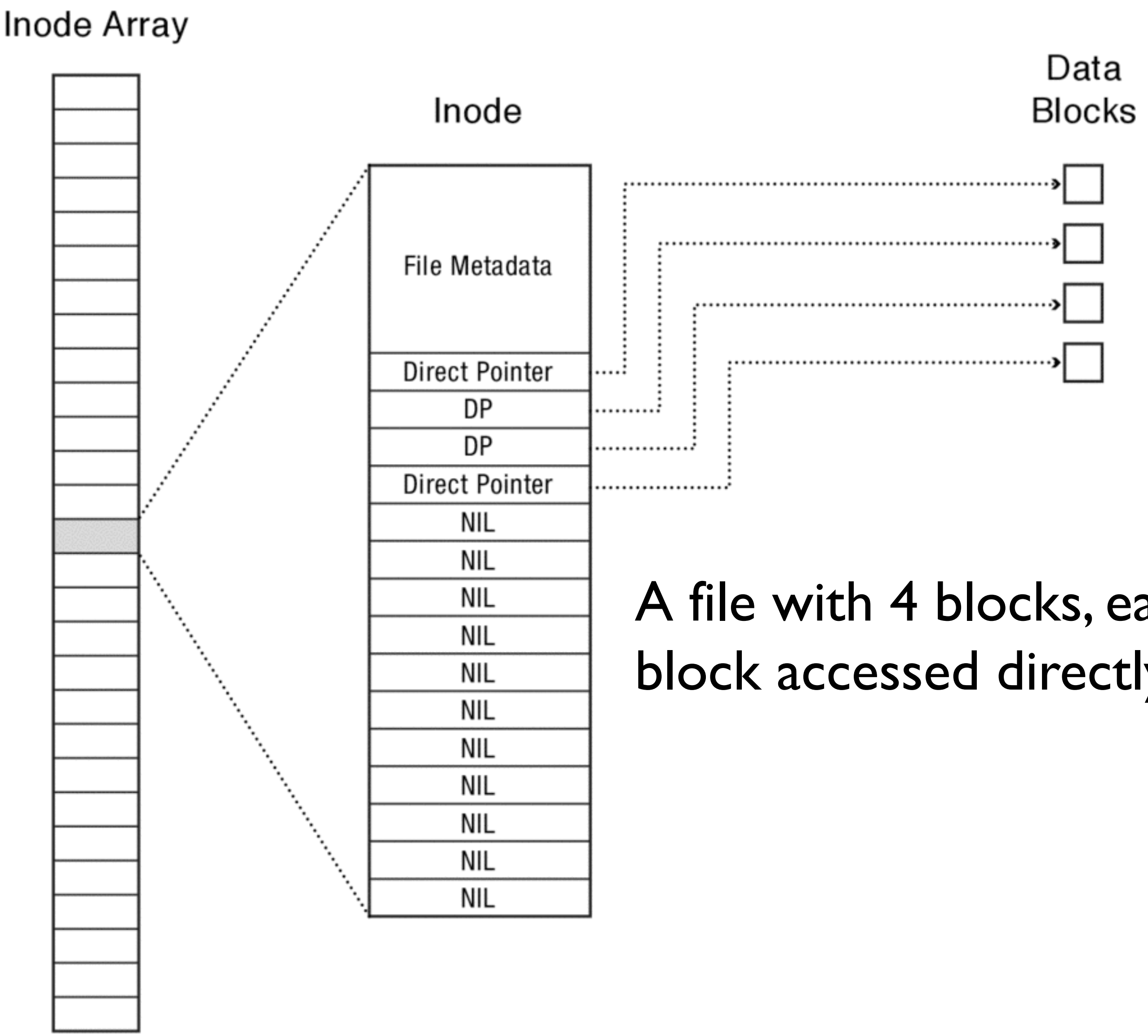
⇒To store a 4B file, how much space we need?

- 4B data + small inode + 3x 4KB indirect blocks
- How about our asymmetric tree?

Asymmetric vs. Symmetric

- In a symmetric tree with eac

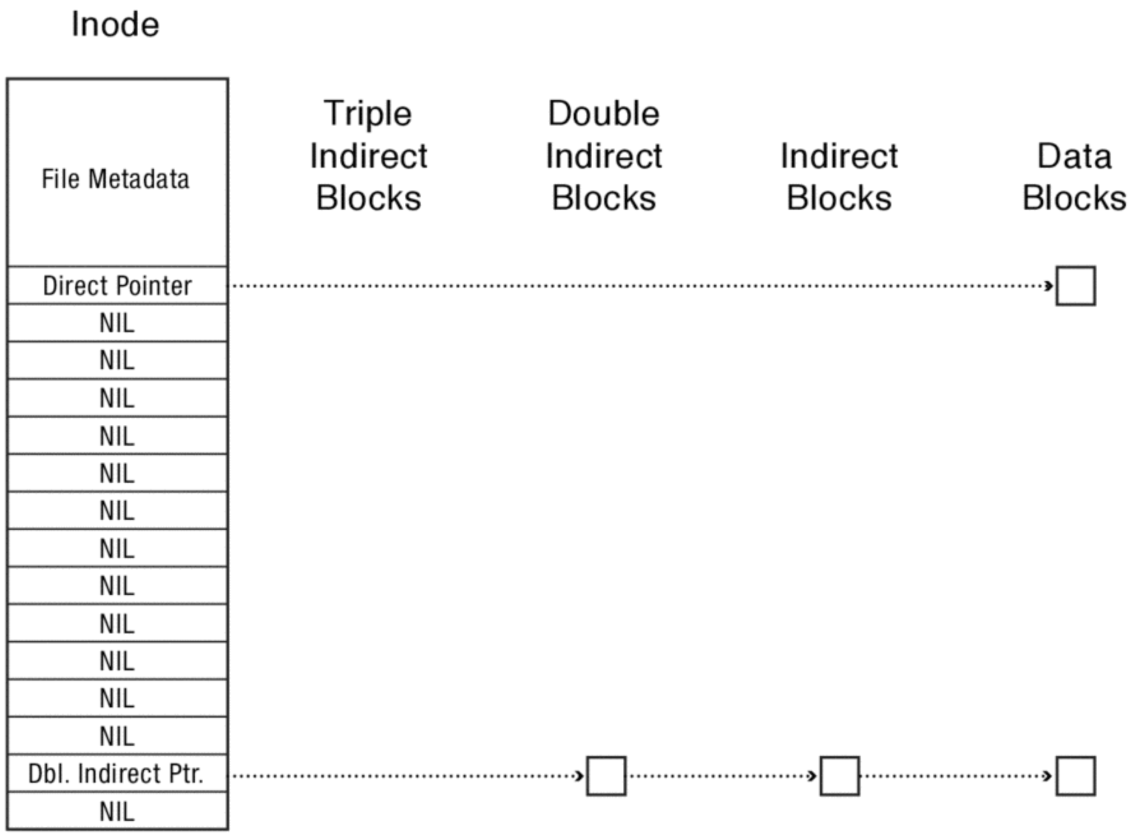
⇒To store a 4B file, how muc
  - 4B data + small inode + 3x 4
  - How about our asymmetric t



A file with 4 blocks, each block accessed directly

Sparse Files

- FFS can support sparse files, in which one or more ranges of empty space are surrounded by file data.
  - Those empty space shall not consume disk space.



<= A sparse file with 4KB data at offset 0, and 4KB data at offset 2^{30}.

Command ls shows it takes 1.1GB.
Command du shows it takes 16KB.

Free Space Management

- FFS allocates a bitmap with one bit per storage block. The i-th bit in the bitmap indicates whether the i-th block is free or in use.
- The position of FFS's bitmap is fixed when the file system is formatted.
  - So it is easy to find the part of the bitmap that identifies free blocks near any location of interest.

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"

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 ⇒ 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 still find many of the files (even if directories disconnected)
- Part of the Fast File System (FFS)
  - General optimization to avoid seeks

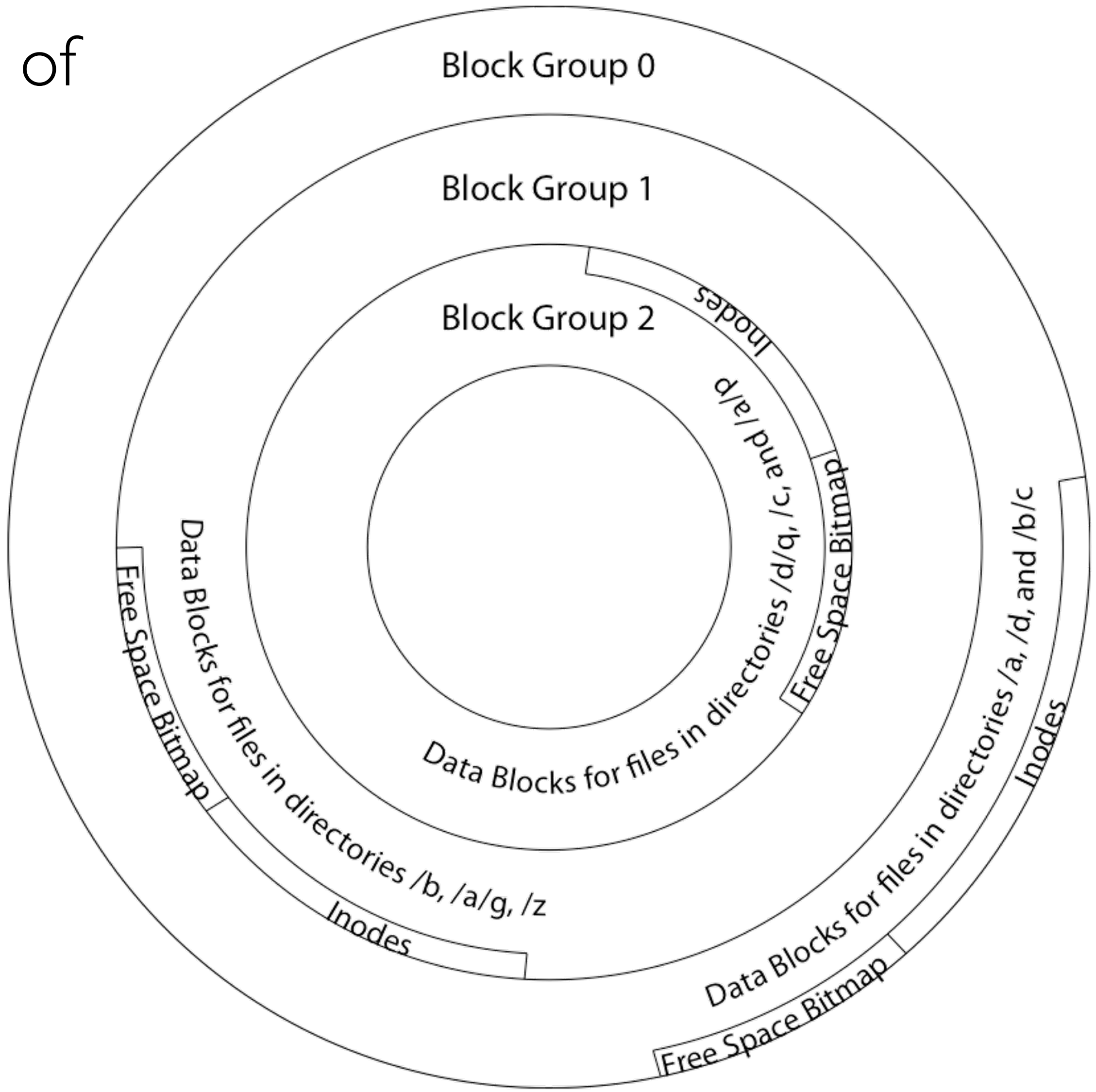
Locality Heuristics

- **Block group placement:** FFS places data to optimize for the common case where a file's data blocks, a file's data and metadata, and different files from the same directory are accessed together.

- **Reserved space:** FFS reserves some fraction of the disk's space (e.g., 10%) and presents a slightly reduced disk size to applications.
  - When disk is full, there's little opportunity for file system to optimize locality.
  - Sacrifices a little disk capacity for better locality thus reduced seek times.

Block Group Placement

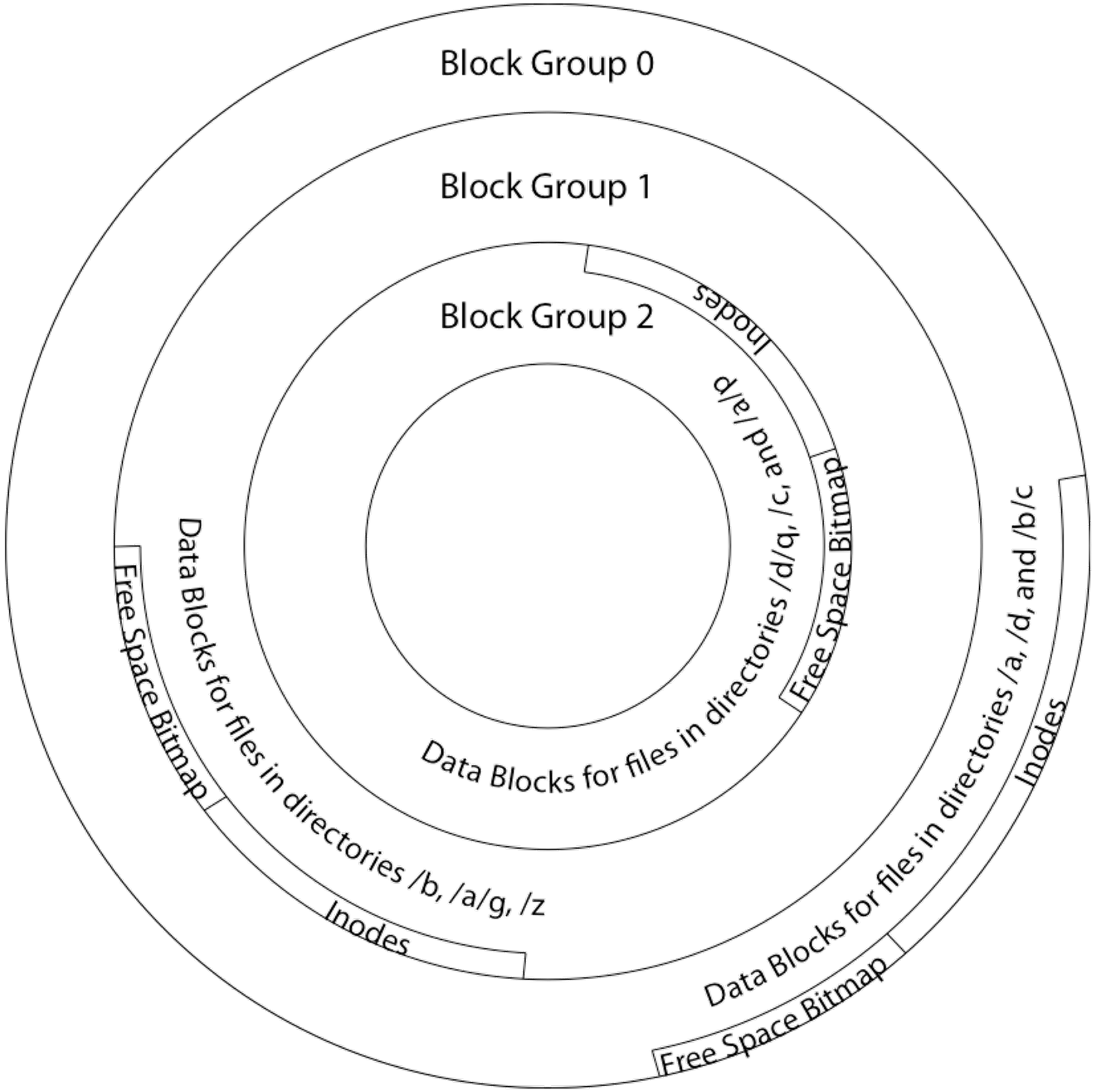
- File system volume is divided into a set of block groups
  - Small seek time
- Data blocks, metadata, and free space are distributed to different block
  - Avoid huge seeks between user data and system structure



Block Group Placement

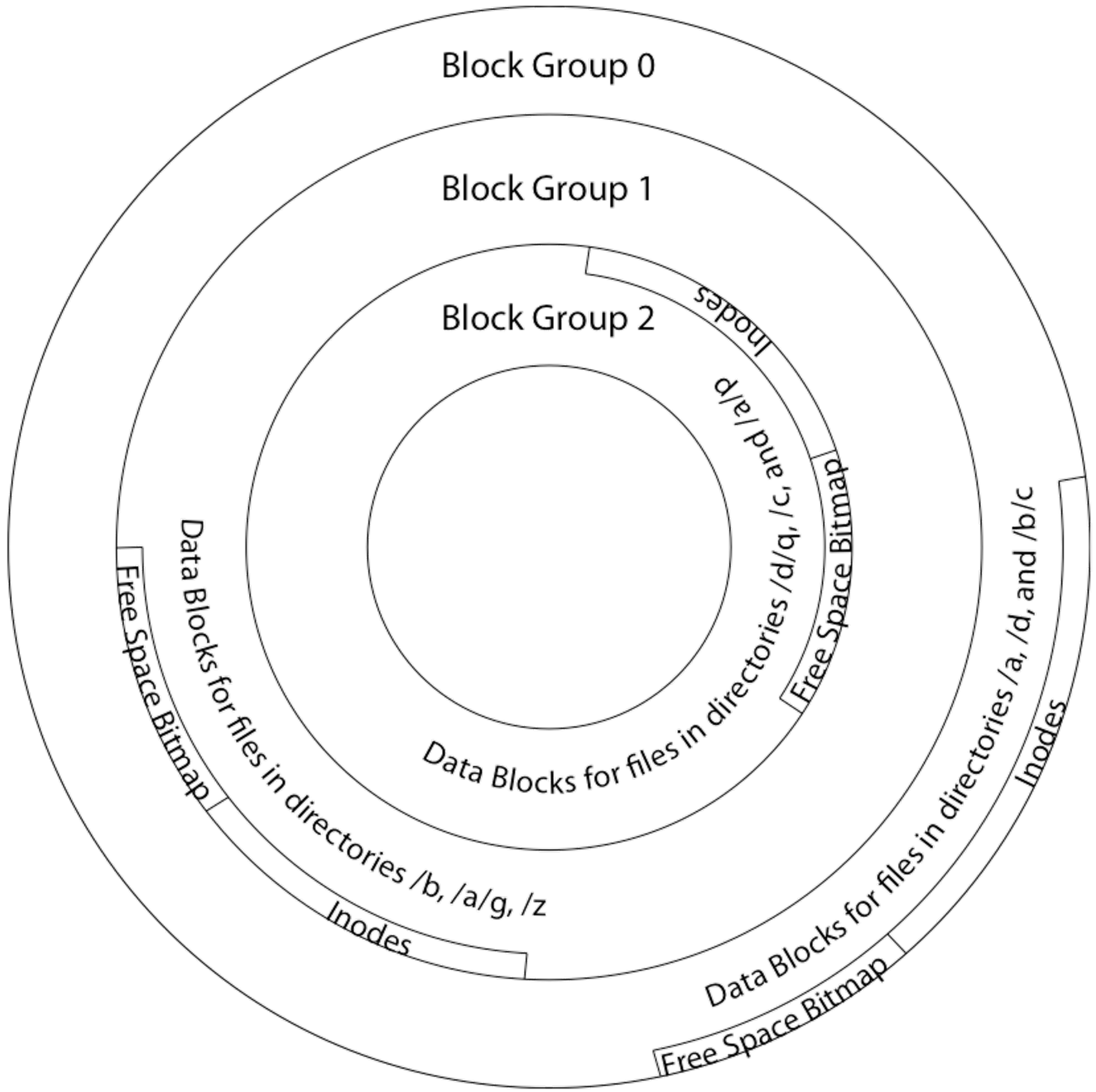
- Files in the same directory are placed in the same block group
  - The same for the "directory file" as well
  - i.e., when a new file is created, find an inode number within the block where its directory resides and give it to the file.
    - ❑ Unless there's no free inode number in that block

- But don't put the directory and its sub-directory together
  - Though they might have locality, it will easily fill the block.

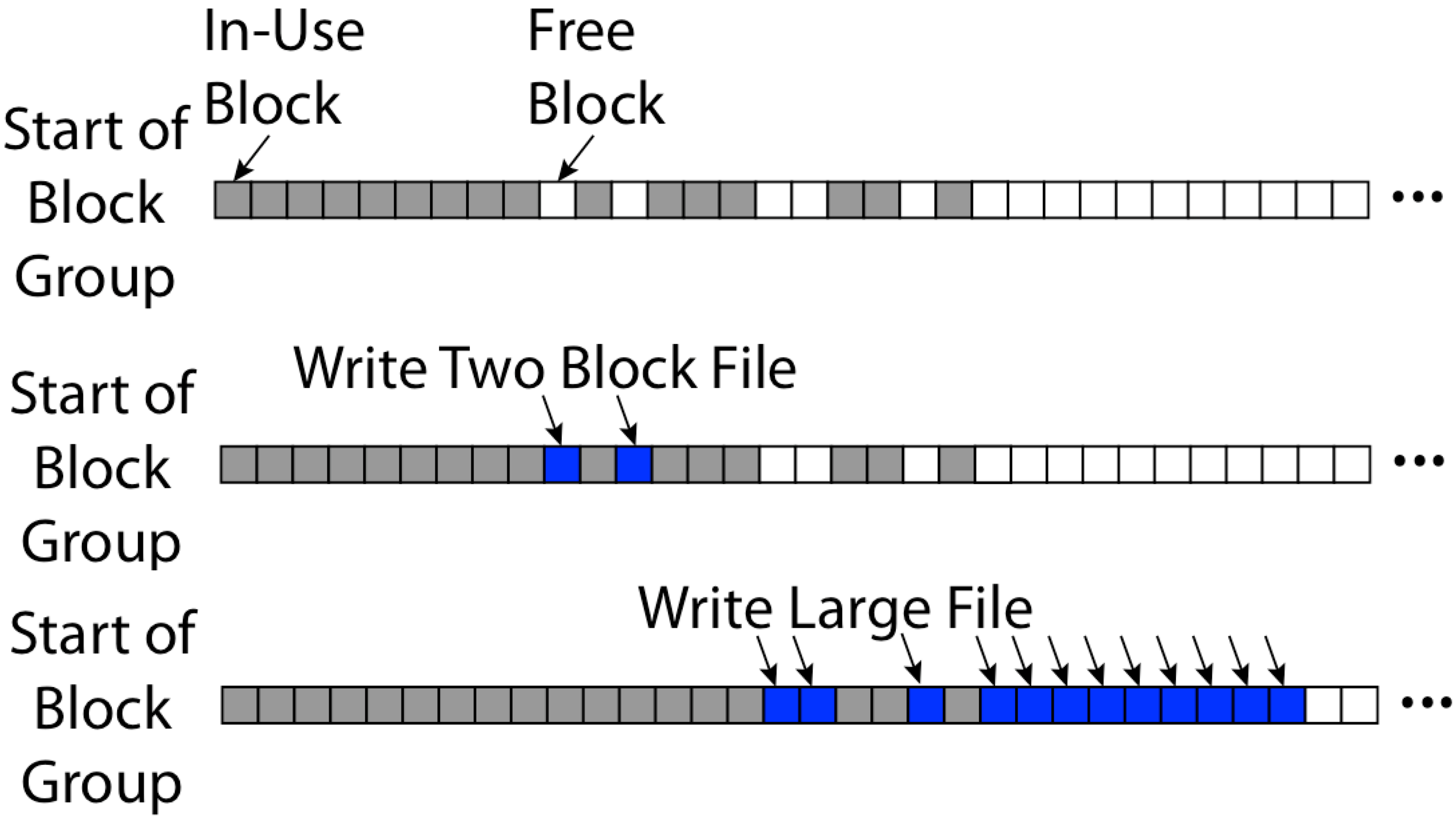


Block Group Placement

- 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



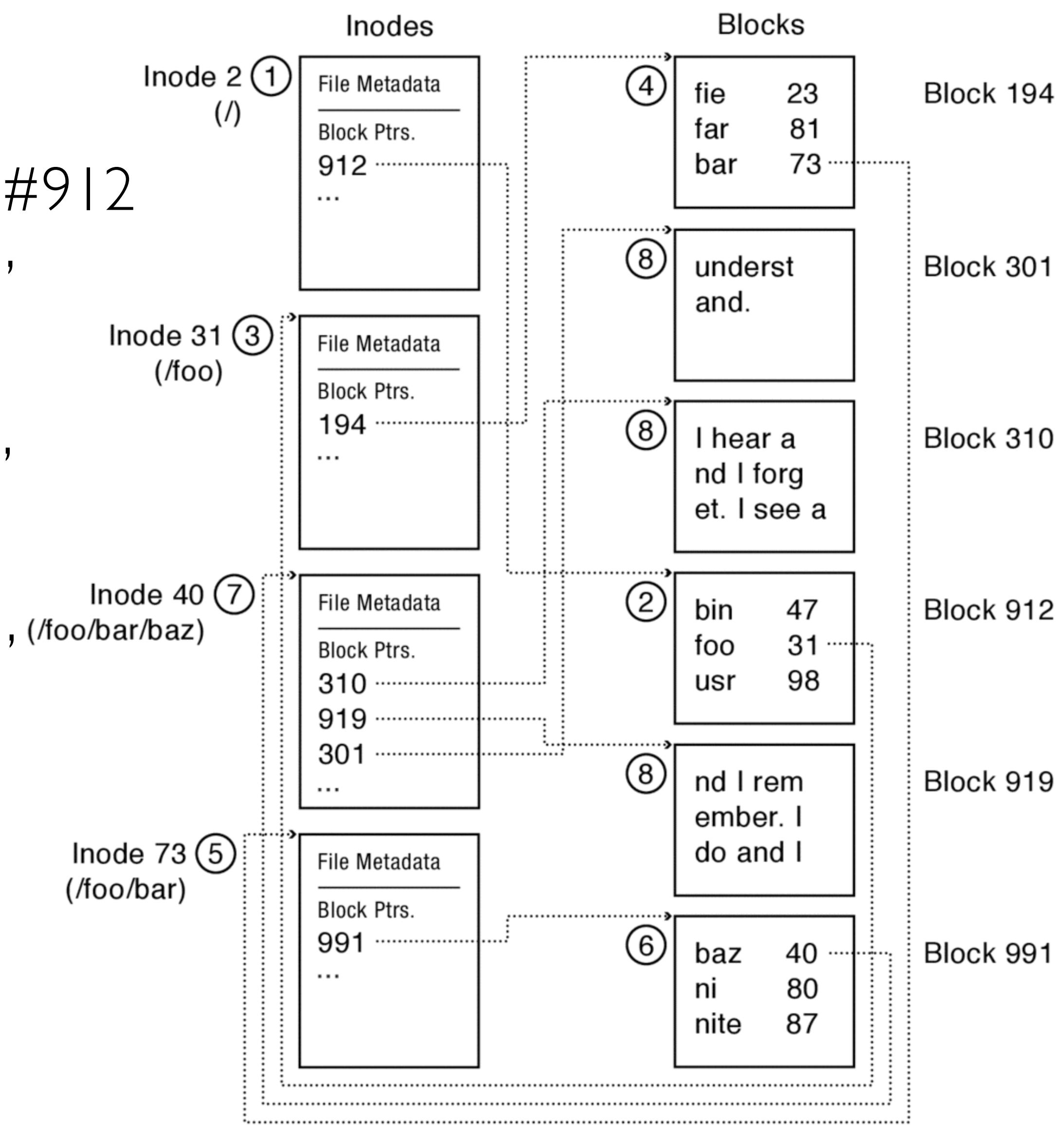
UNIX 4.2 BSD FFS First Fit Block Allocation



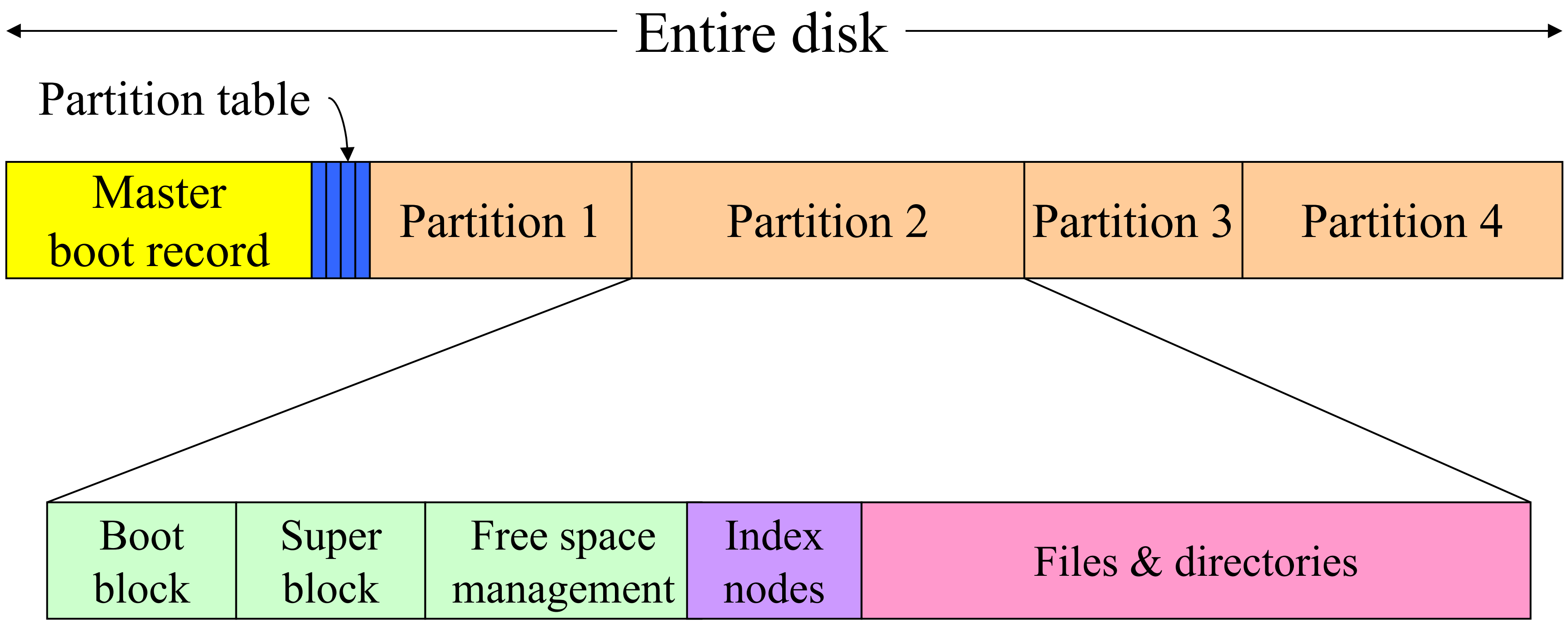
Put All Things Together (FFS)

Example: read the file /foo/bar/baz

1. Read "/" root inode #2's inode, get block #912
2. From block #912, get inode #31 for "foo"
3. From inode #31, get block #194
4. From block #194, get inode #73 for "bar"
5. From inode #73, get block #991
6. From block #991, get inode #40 for "baz"
7. From inode #40, get 3 data blocks
   - Block 310
   - Block 919
   - Block 301



Carving Up the Disk



- Boot block: the initial bootstrap program to load OS
- Super block: describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, inode number of the root directory, magic number, etc

FFS Summary

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

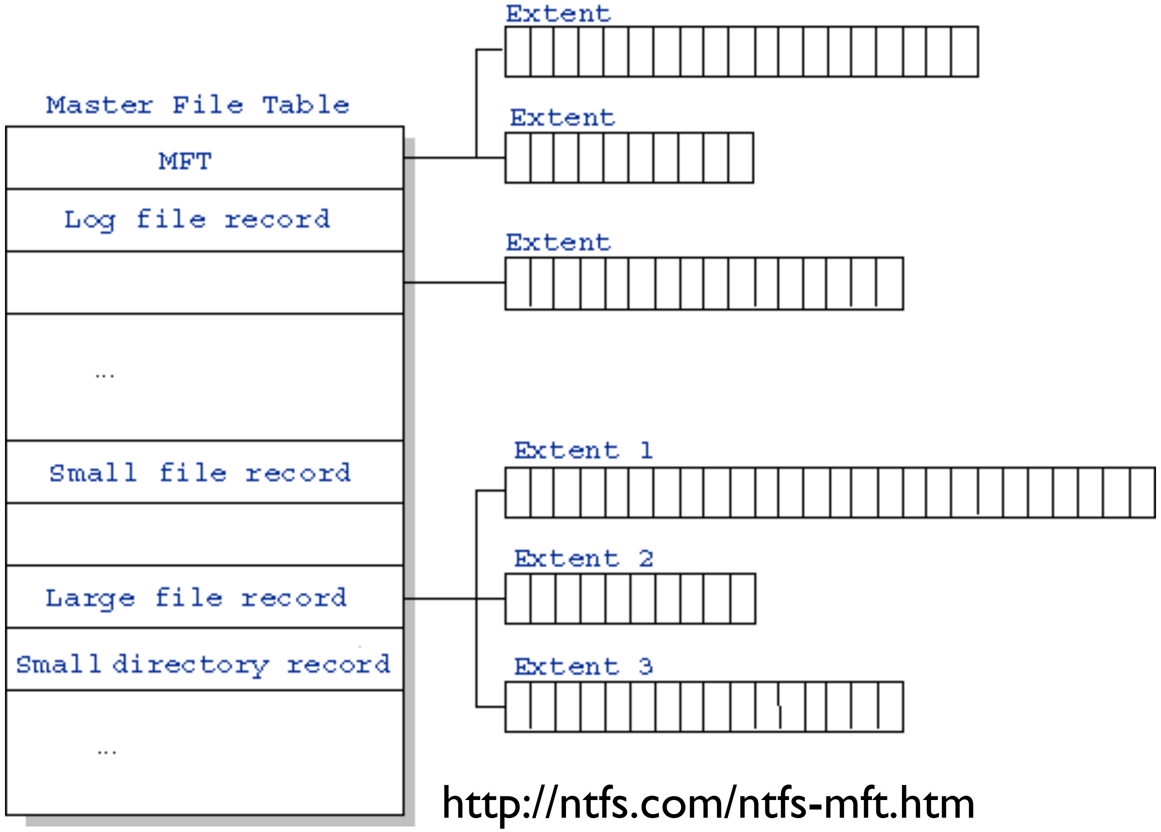
- Cons
  - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on disk
  - Need to reserve 10-20% of free space to prevent fragmentation

NTFS

- New Technology File System (NTFS)
  - Default on Microsoft Windows systems
- Variable length extents
  - Rather than fixed blocks
- Everything (almost) is a sequence of <attribute (属性):value> pairs
  - Meta-data and data
- Mix direct and indirect freely
- Directories organized in B-tree structure by default

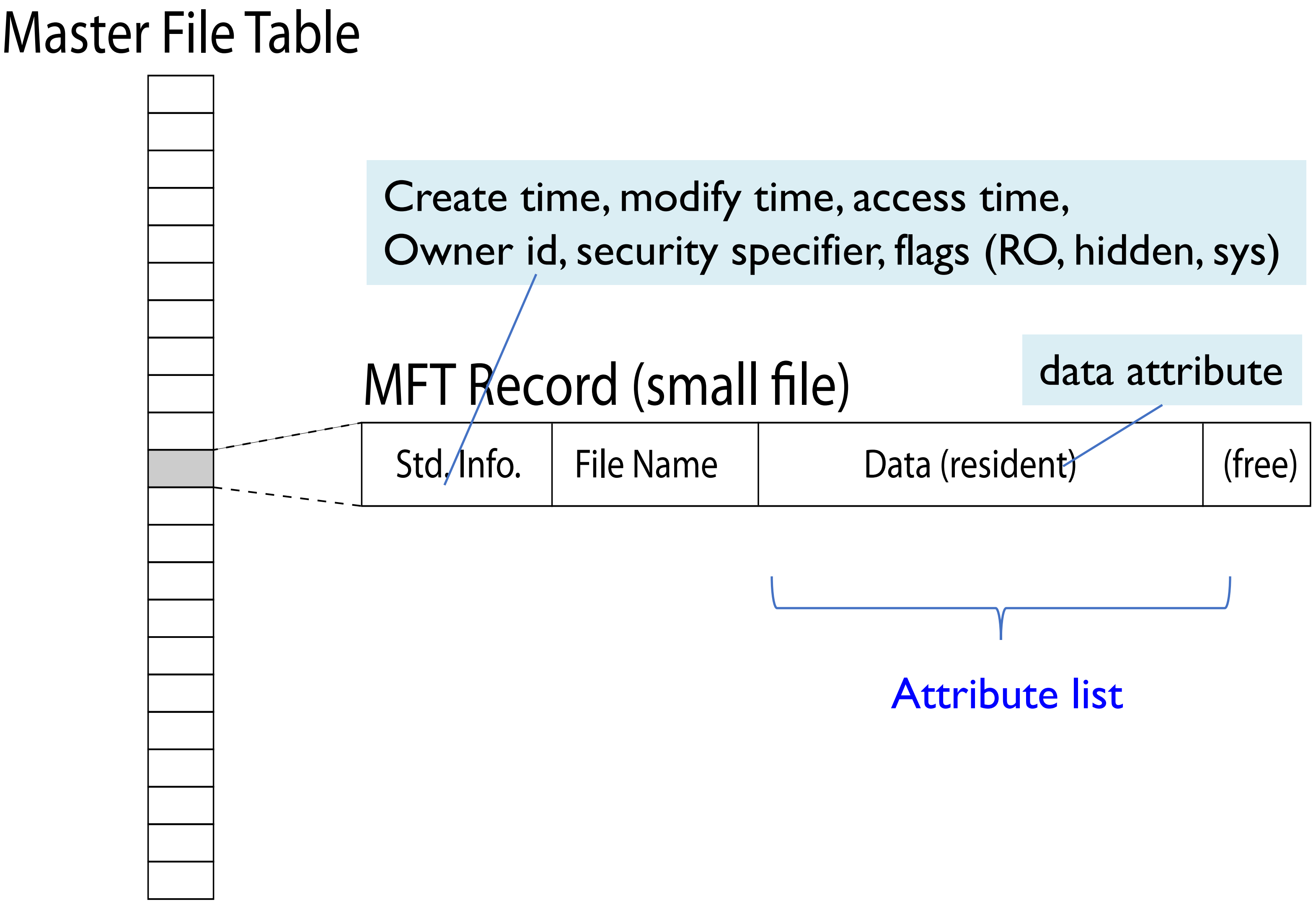
NTFS

- 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 attribute, 非常驻属性)
- 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

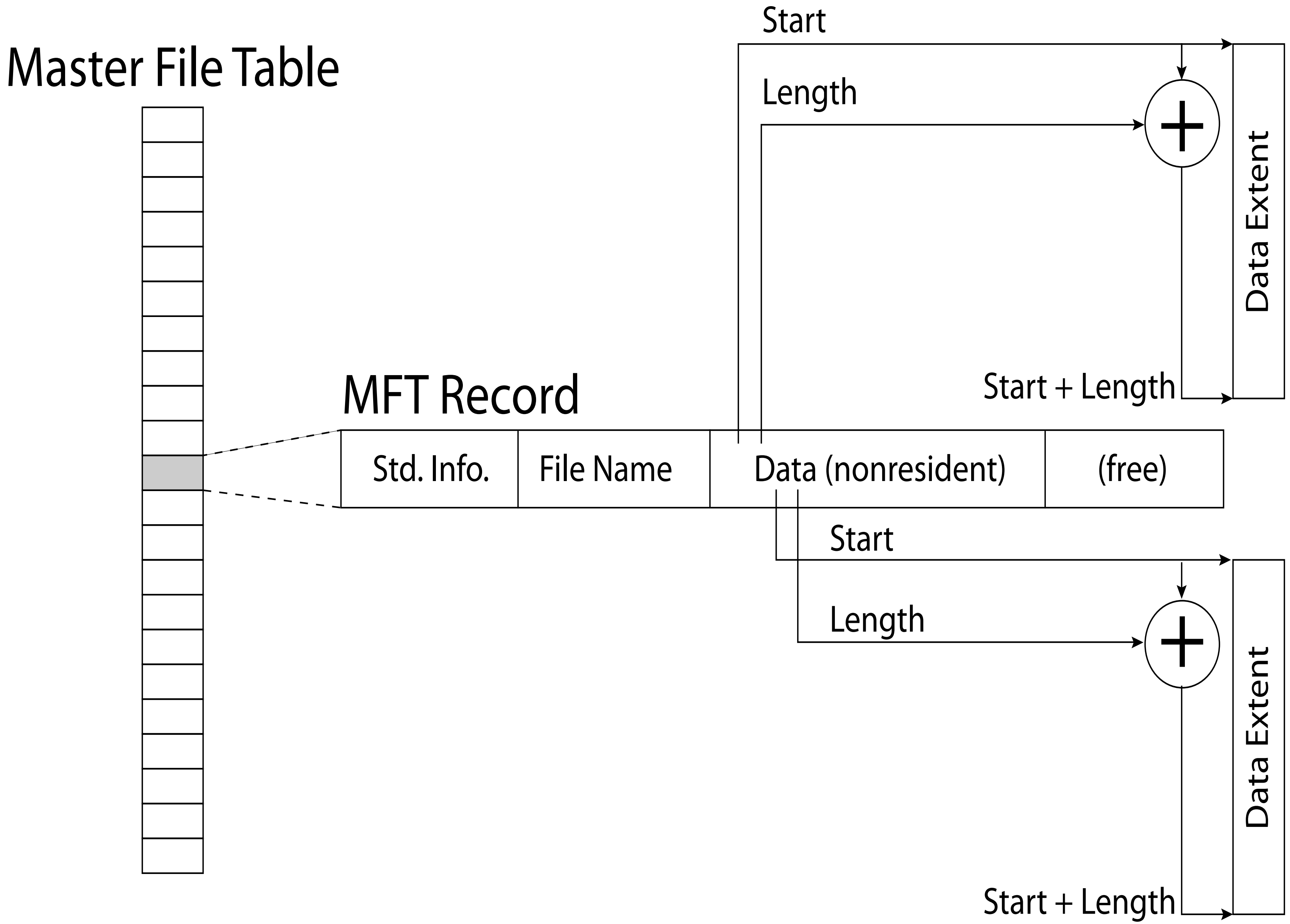


http://ntfs.com/ntfs-mft.htm

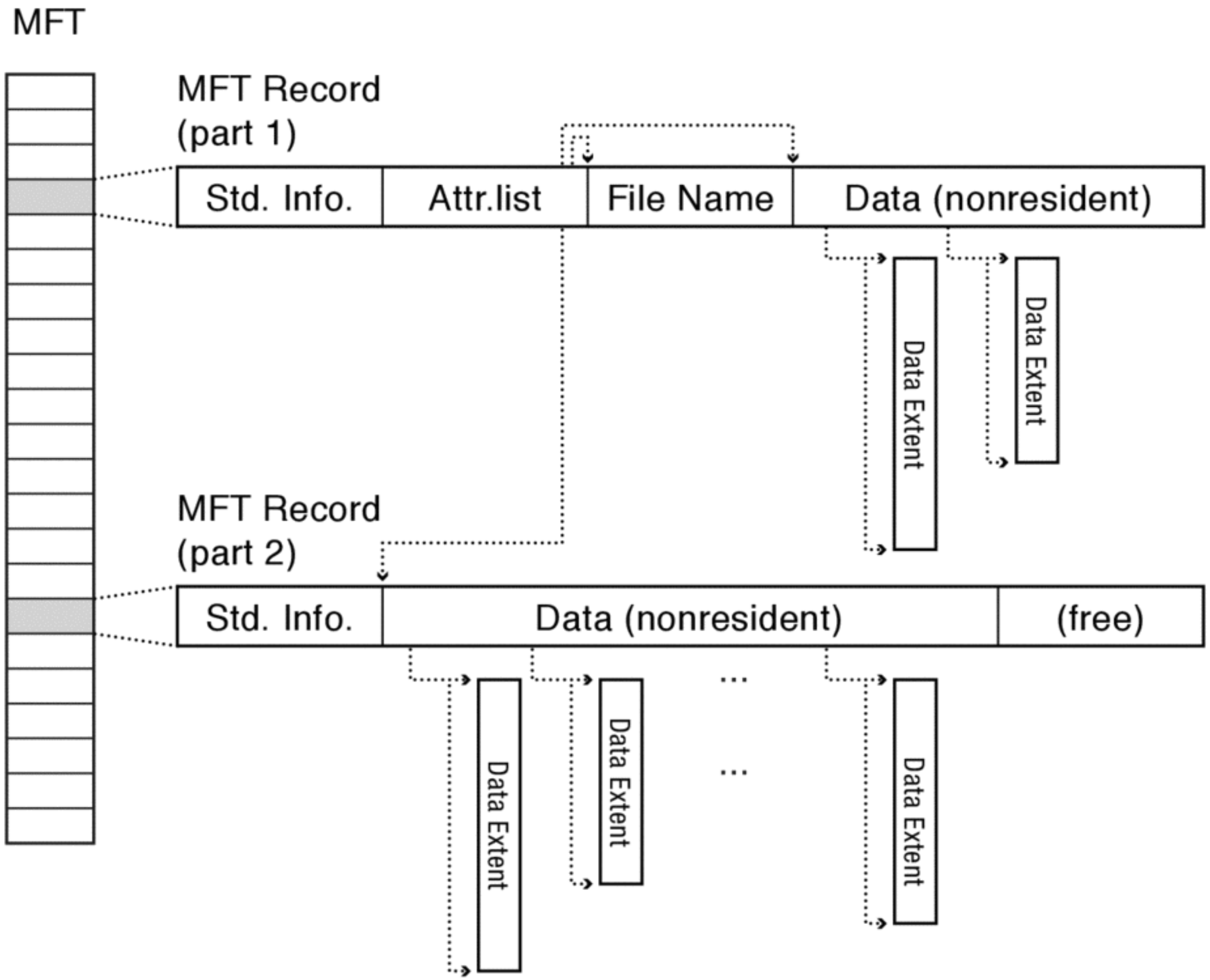
NTFS Small File



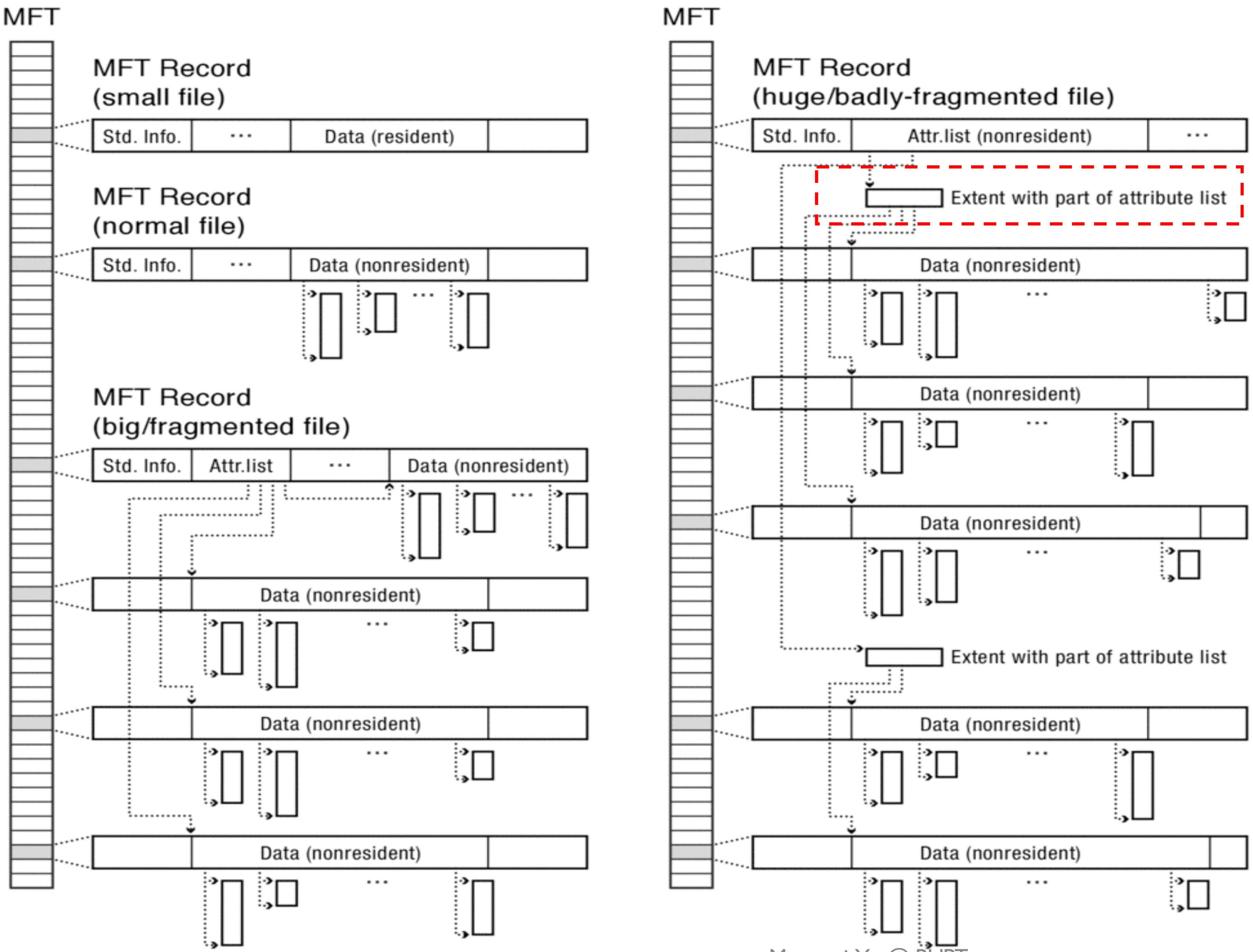
NTFS Medium File



NTFS Large/Fragmented File



NTFS Multiple Indirect Blocks



Even the attribute list becomes nonresident!

Why it is possible??

NTFS Details

- File system metadata is stored in files with well-known low-numbered file numbers
  - File number 0 ($MFT) is the MFT itself
  - File number 5 is the root directory
  - File number 6 is the free space bitmap
  - File number 8 contains a list of the volume's bad blocks
  - File number 9, called $Secure, contains security and access control information.

- If MFT is stored as a file, how do we read it..?
  - To locate the MFT, the first sector of an NTFS volume includes a pointer to the first entry (why?) of the MFT.

NTFS Locality Heuristics

- Best fit: where the system tries to place a newly allocated file in the smallest free region that is large enough to hold it.
  - In most implementations
- An important NTFS feature: SetEndOfFile() to specify the expected size of a file at creation time.
  - Why it is useful?
- To avoid having $MFT become fragmented, NTFS often reserves part of the disk (e.g., the first 12.5% of the volume) for MFT expansion
  - Why we didn't care about fragmentation in FFS? (doesn't mean there's no fragmentation in FFS! For example, still internal fragmentation)
  - Recall: segmenting vs. paging

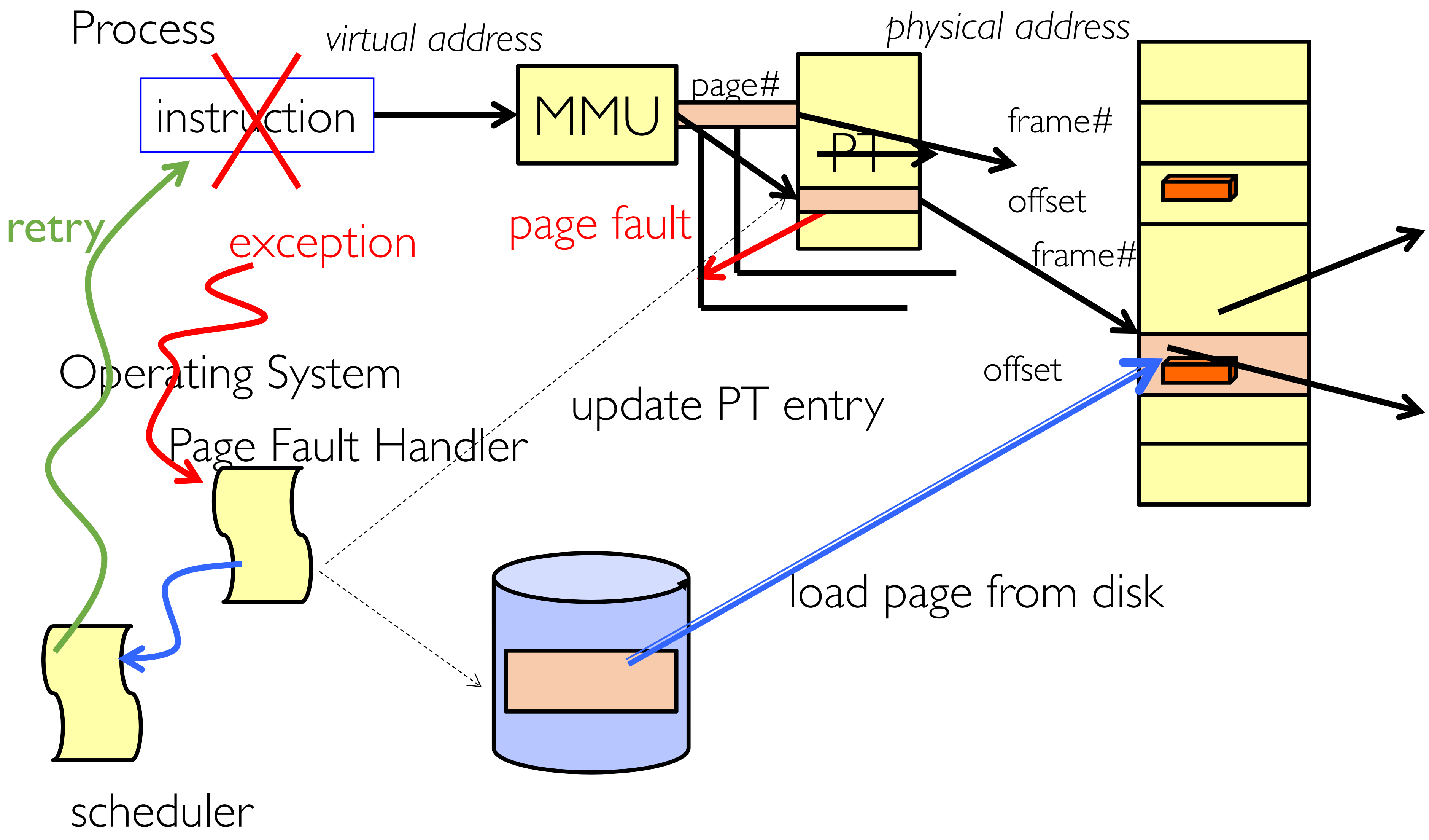
Memory Mapped Files

- 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

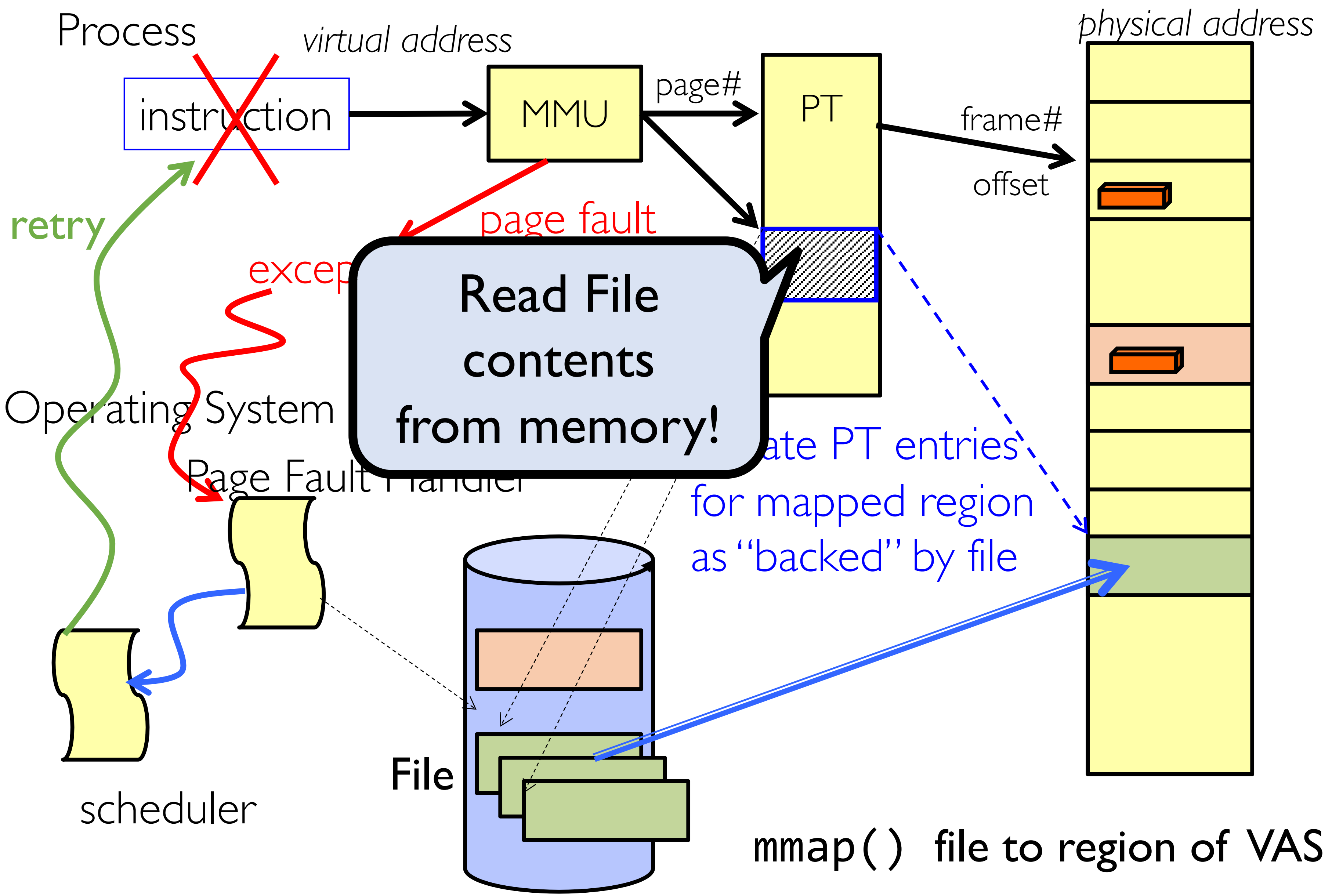
- What if we could "map" the file directly into an empty region of our address space
  - Implicitly "page it in" when we read it
  - Write it and "eventually" page it out

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

Recall: Who Does What, When?



Using Paging to `mmap()` Files



`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

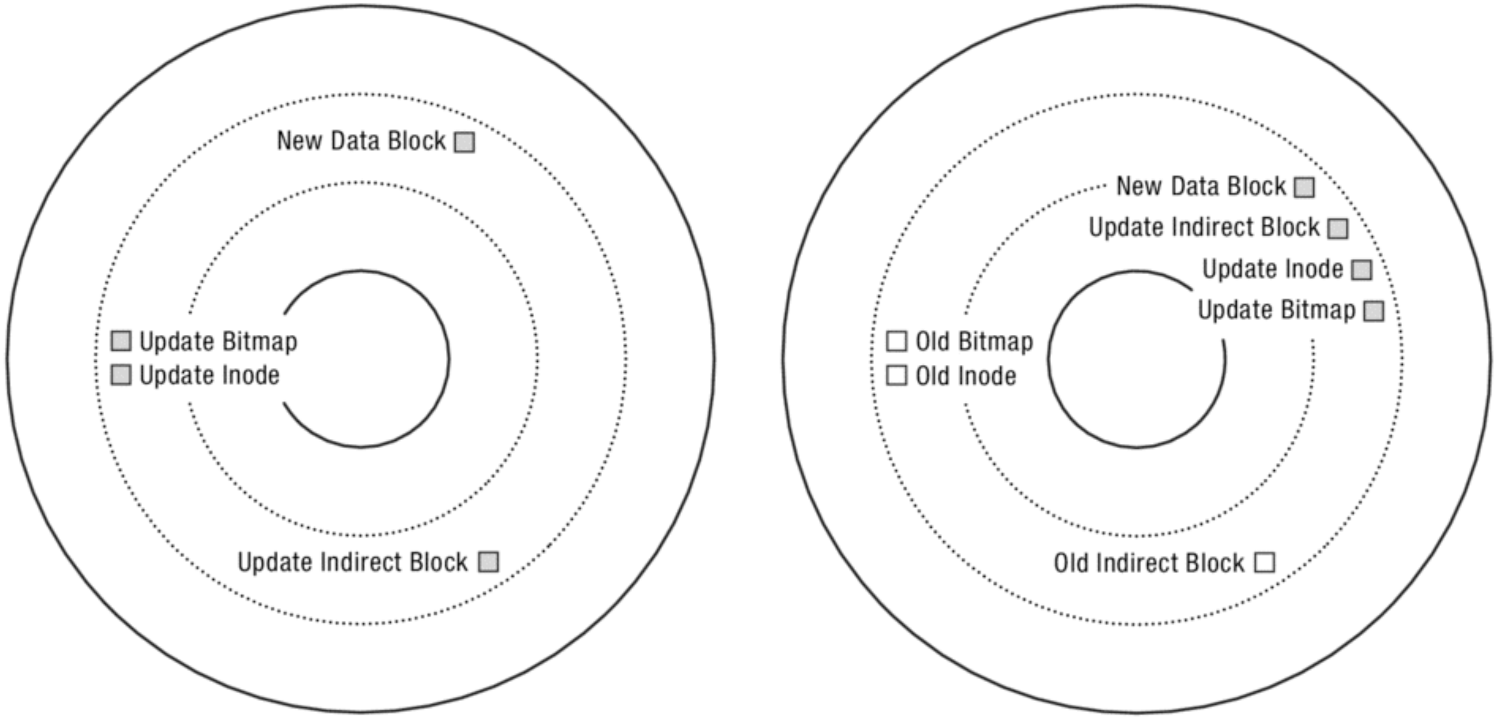
```
#include <sys/mman.h> /* also stdio.h, stdlib.h, string.h, fcntl.h, unistd.h */

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|P
  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
```

Other File Systems..

- Copy-on-write (COW) file system: when updating an existing file, it does not overwrite the existing data or metadata; instead, it writes new versions to new locations
  - Turning random I/O updates to sequential ones.



Read textbook for more information!

Goals for Today

- Directories: naming data
  - How do we convert a file name to the file number?
- Files: finding data
  - How do we locate storage block based on file number?
- Virtual file systems (VFS)
  - How do we make different FSs work together easily?

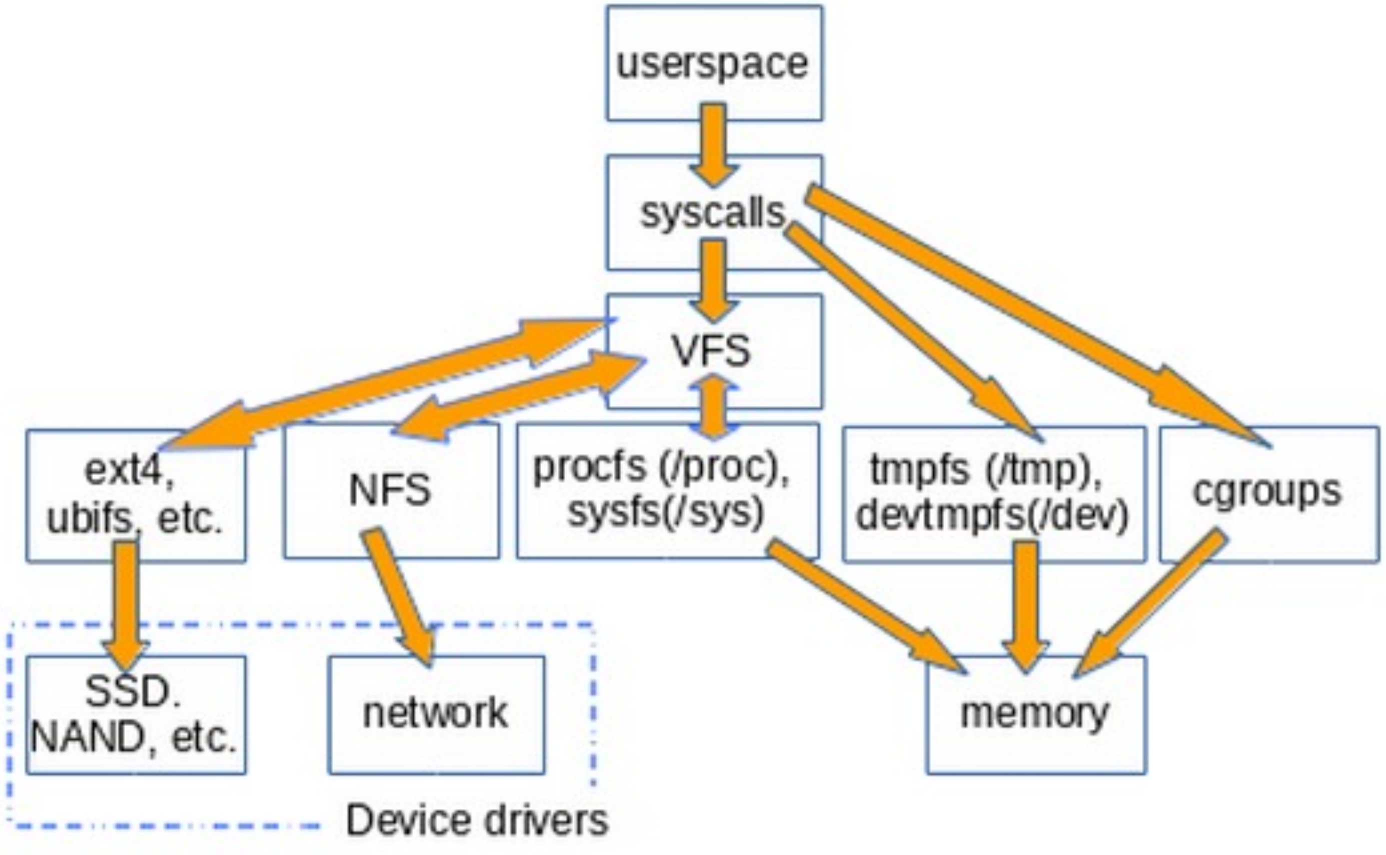
file name offset —directory→ *file number offset* —index structure→ *Storage block*

History

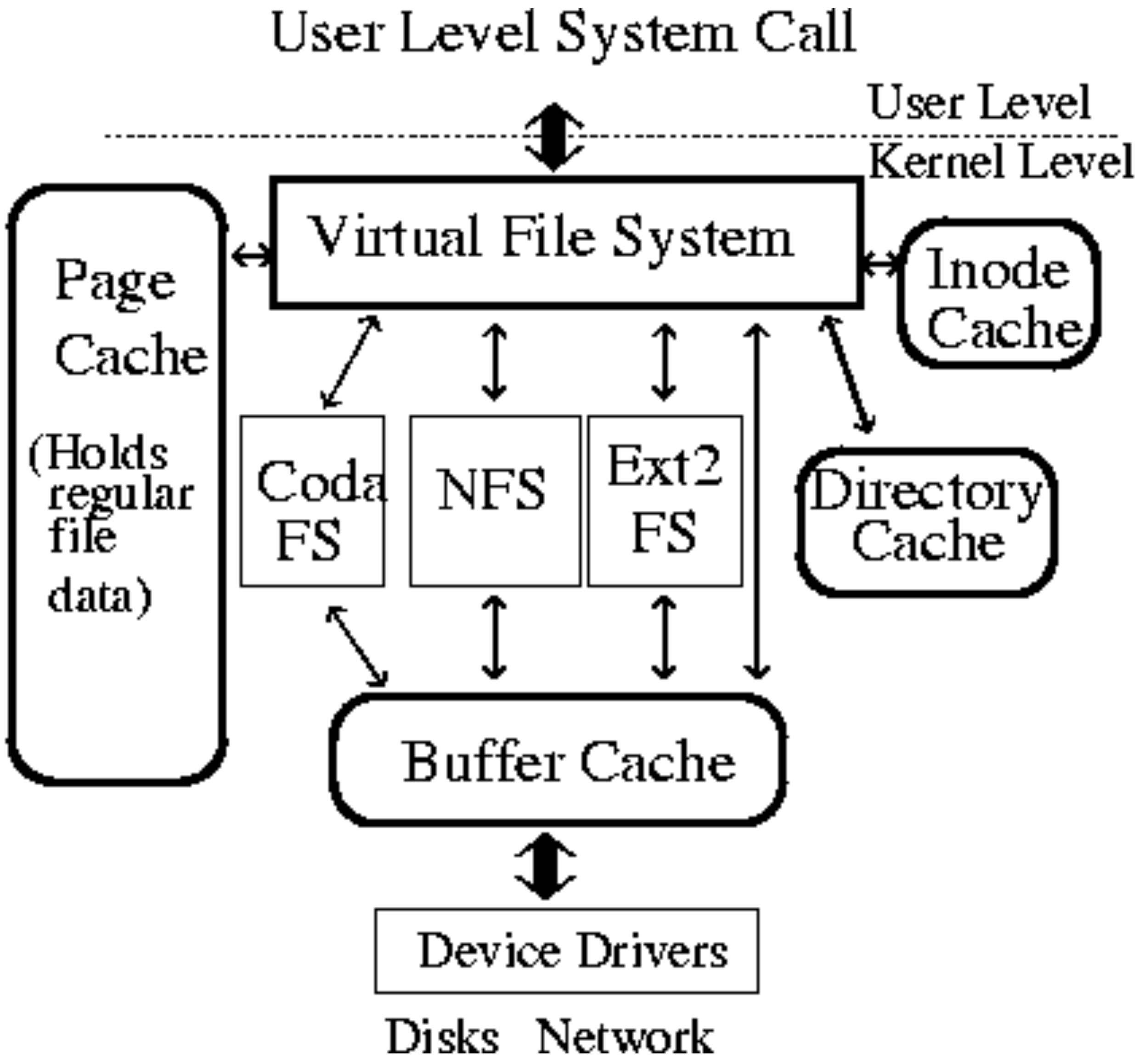
- Early OSes provided a single file system
  - In general, system was tailored to target hardware

- People became interested in supporting more than one file system type on a single system
  - Any guesses why?
  - Networked file systems: sharing parts of a file system across a network of workstations

Virtual File System (VFS)



Virtual File System (VFS)



Modern VFS

- Dozens of supported file systems
  - Allows new features and designs transparent to apps
  - Interoperability with removable media and other OSes
- Independent layer from backing storage
  - In-memory file systems (ramdisks)
  - Pseudo file systems used for configuration
    - (/proc, /devtmps…) only backed by kernel data structures
- And, of course, networked file system support

What the VFS Does

- The VFS is a substantial piece of code
  - not just an API wrapper

- Caches file system metadata (e.g., names, attributes)
  - Coordinates data caching with the page cache
- Enforces a common access control model
- Implements complex, common routines
  - Path lookup
  - Opening files
  - File handle management

User's Perspective

- Single programming interface
  - (POSIX file system calls – open, read, write, etc.)
- Single file system tree
  - Remote FS can be transparently mounted (e.g., at /home)
- Alternative: Custom library for each file system
  - Much more trouble for the programmer

FS Developer's Perspective

- FS developer responsible for implementing standard objects/functions called by the VFS
  - Primarily populating in-memory objects
    - ❑Typically from stable storage
  - Sometimes writing them back
- Can use block device interfaces to schedule disk I/O
  - And page cache functions
  - And some VFS helpers
- Analogous to implementing Java abstract classes

High-level FS dev. tasks

- Translate between VFS objects and backing storage (whether device, remote system, or other/none)
  - Potentially includes requesting I/O
- Read and write file pages

- VFS doesn't prescribe all aspects of FS design
  - More of a lowest common denominator
- Opportunities: (to name a few)
  - More optimal media usage/scheduling
  - Varying on-disk consistency guarantees
  - Features (e.g., encryption, virus scanning, snapshotting)

Core VFS Abstractions

- **super block:** FS-global data
  - Early/many file systems put this as first block of partition
- **inode:** (index node): metadata for one file
- **dentry:** (directory entry): name to inode mapping
- **file object:** pointer to dentry and cursor (file offset)

- SB and inodes are extended by file system developer

Core VFS Abstractions

```
struct super_block {
        struct list_head                s_list;    /* Keep this first */
        ……
        unsigned long                   s_blocksize;
        ……
        unsigned long long              s_maxbytes; /* Max file size */
        struct file_system_type         *s_type;
        struct super_operations         *s_op;
        ……
        struct dentry                   *s_root;
        ……
        struct list_head                s_dirty; /* dirty inodes */
        ……
        union {
                struct minix_sb_info    minix_sb;
                struct ext2_sb_info     ext2_sb;
                struct ext3_sb_info     ext3_sb;
                struct ntfs_sb_info     ntfs_sb;
                struct msdos_sb_info    msdos_sb;
                ……
                void                    *generic_sbp;
        } u;
        ……
};
```

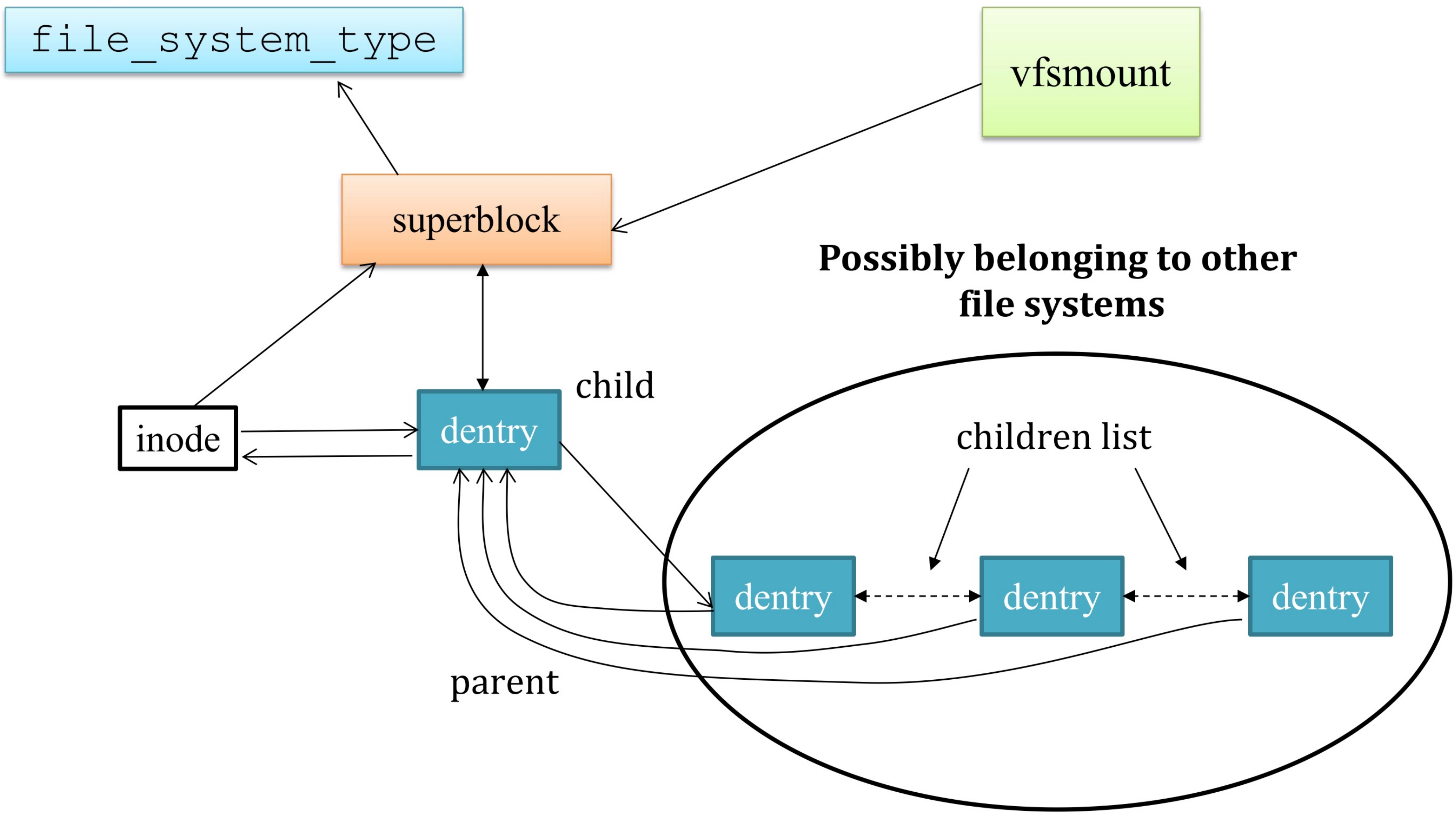
Core VFS Abstractions

```
struct dentry {
        unsigned int dflags;
        ......
        struct inode  * d_inode;   /* Where the name belongs to */
        struct dentry * d_parent;  /* parent directory */
        struct list_head d_hash;   /* lookup hash list */
        ......
        struct list_head d_child;  /* child of parent list */
        struct list_head d_subdirs;       /* our children */
        ......
        struct qstr d_name;
        ......
        struct lockref d_lockref;  /*per-dentry lock and refcount*/
        struct dentry_operations  *d_op;
        struct super_block * d_sb; /* The root of the dentry tree*/
        ......
        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

Core VFS Abstractions

```
struct inode {
        ……
        struct list_head        i_dentry;
        ……
        uid_t                   i_uid;
        gid_t                   i_gid;
        ……
        unsigned long           i_blksize;
        unsigned long           i_blocks;
        ……
        struct inode_operations     *i_op;
        struct file_operations      *i_fop;
        struct super_block          *i_sb;
        wait_queue_head_t           i_wait;
        ……
        union {
                ……
                struct ext2_inode_info      ext2_i;
                struct ext3_inode_info      ext3_i;
                ……
                struct socket               socket_i;
                ……
                void                        *generic_ip;
        } u;
};
```

VFS Global Organization



Embedded Inodes

- Many FSes embed VFS inode in FS-specific inode

```
struct myfs_inode {
        int ondisk_blocks[];
        /* other stuff*/
        struct inode vfs_inode;
}
```

- Why? Finding the low-level from inode is simple
  - Compiler translates references to simple math

https://compas.cs.stonybrook.edu/~nhonarmand/courses/fa14/cse506.2/slides/vfs.pdf

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
- Deep interactions between mem management, file system, sharing
  - `mmap()`: map file or anonymous segment to memory