Operating System Design and Implementation

Lecture 19: File System

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

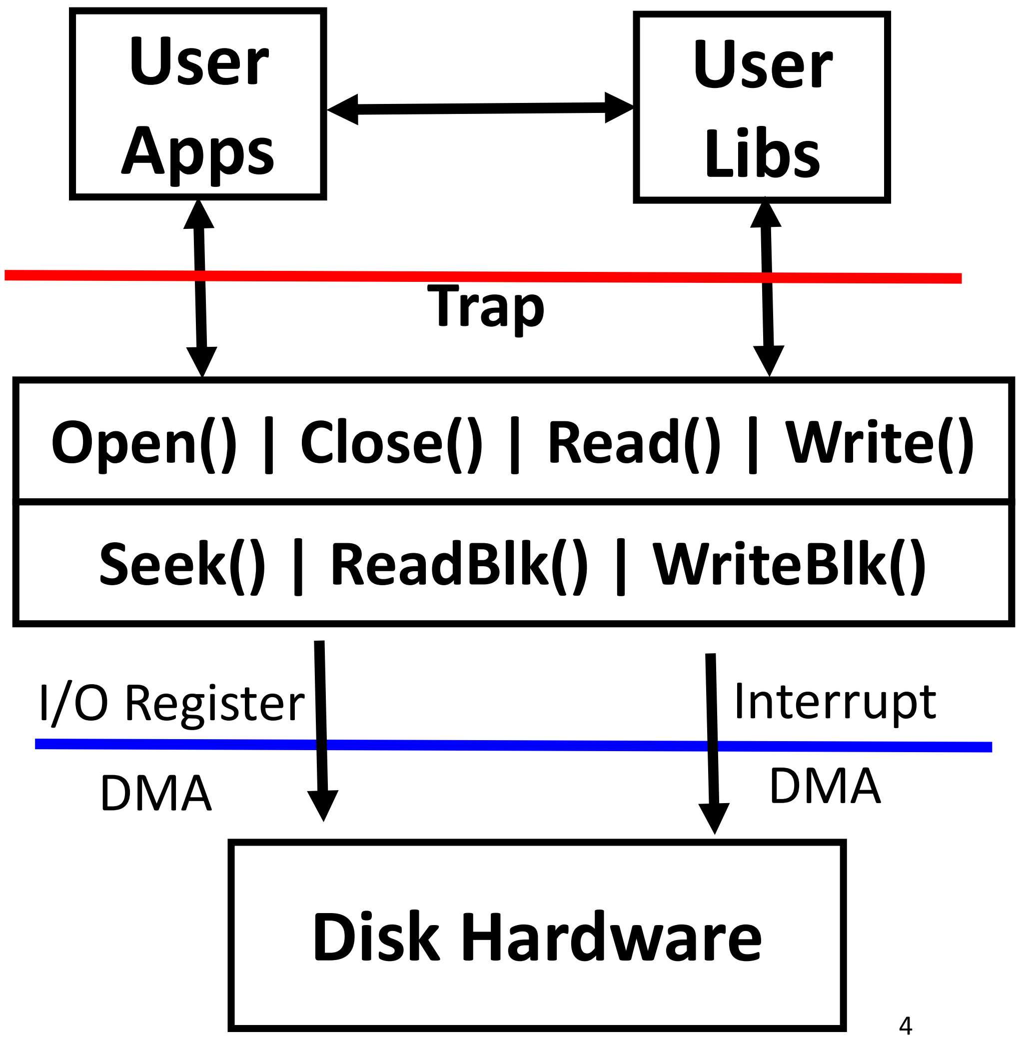
- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University
- CSE 506, operating system, 2016,
  https://www.cs.unc.edu/~porter/courses/cse506/s16/slides/sync.pdf

Outline

- File system structures
  - Inode
  - Superblock …
- Allocating data blocks
  - Link file allocation
  - Index file allocation
  - Multi-level indexed file allocation
- Soft vs. hard link
- File I/O operations

File system layers

- **User's viewpoint**
  - **Objects:** files, directories, bytes
  - **Operations**: create, read, write delete, rename, move, seek
- **Physical viewpoint**
  - **Objects**: sectors, tracks, disks
  - **Operations**: seek, R/W block
- **User <-> OS layer**
  - User library hides many details
  - OS can directly R/W user data
- **OS <-> Hardware**
  - I/O registers, interrupts, DMA



What do file system users need ?

- **Persistence**
  - Disk provides basic non-volatile storage
  - OS can enhance persistence via redundancy
- **Speed: Fast access to data**
  - Handle random access efficiently
  - OS can enhance performance via file caching
- **Size: can store lots of data**
- **Sharing/protection (access control)**
- **Ease of use**
  - Basic file abstraction (names, offsets, byte streams, …)
  - Directories simplify naming and lookup

File system abstractions

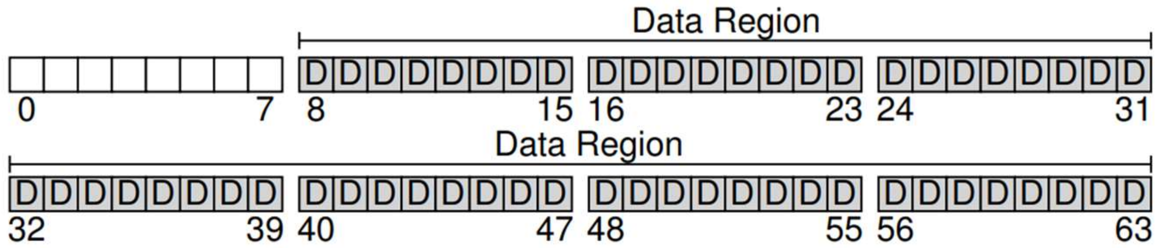
- **File**
  - Basic container of persistent data
- **Directory system**
  - Hierarchical naming relationships
  - Directories are special "files" that index other files
- **Common file access patterns**
  - **Sequential:** data processed in order, byte/record at a time
    - Example: compiler reads a source file
  - **Random access:** address blocks of data based on file offset
    - Example: database searches
  - **Keyed access:** address blocks based on "key" values
    - Example: accessing hash table implemented by key-value

Common file system operations

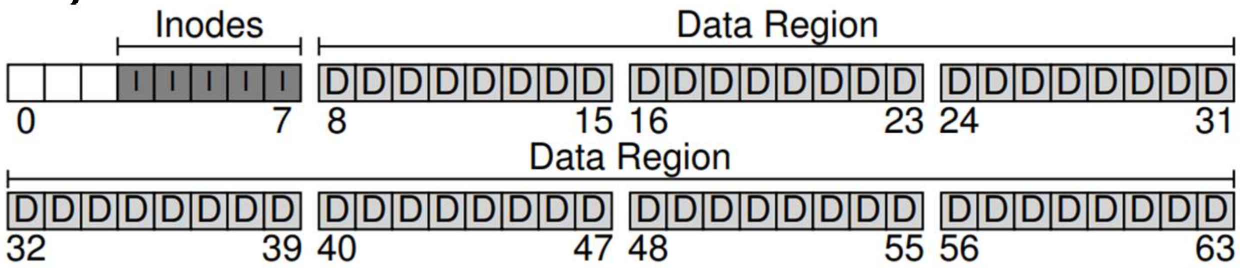
- **Data operations**
  - Create()
  - Delete()
  - Open()
  - Close()
  - Read()
  - Write()
  - Seek()
- **Naming operations**
  - HardLink()
  - SoftLink()
  - Rename()
- **Attribute operations**
  - SetAttribute()
  - GetAttribute()

Attributes include owner, protection, last accessed

File system organization

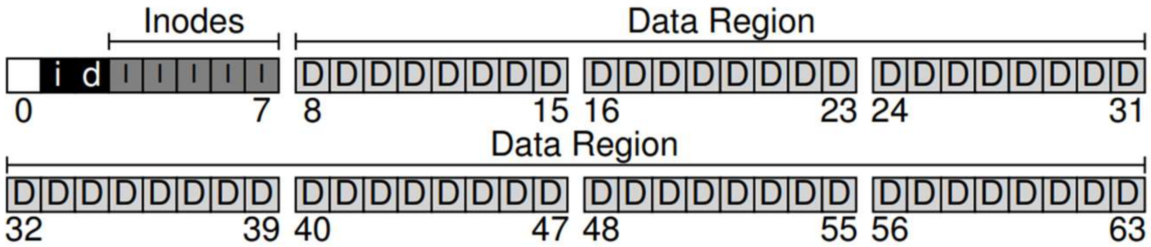


- **Blocks**
  - Divide the disk into data blocks with commonly-used size of 4KB
- **Inode**
  - The metadata of a file such as the size, access rights, modify time etc.
  - Inode tables – holds an array of on-disk inodes
  - E.g. we use 5 out of 64 blocks for inodes
  - An inode is commonly 128 or 256 bytes



https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

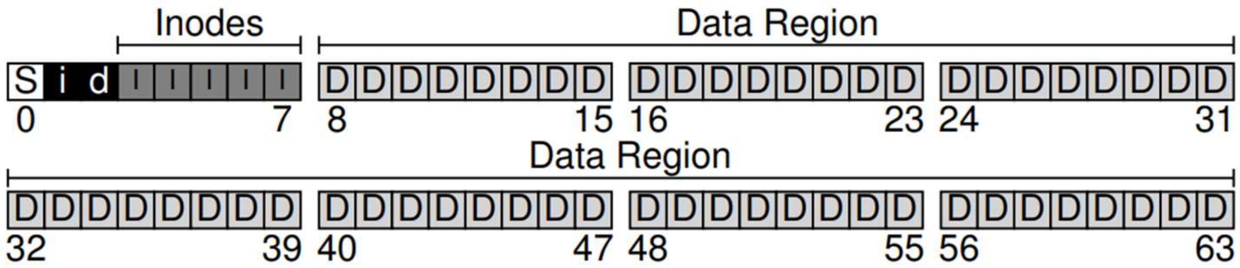
File system organization



- **Inode**
  - Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes, and 80 inodes in this diagram
  - **The number of inode denotes the maximum number of files we can have in a file system**
- **Allocation structures (bitmap)**
  - **Tracking whether inodes or data blocks are free or allocated**
  - Data bitmap (for the data region)
  - Inode bitmap (one for the inode table)
  - Each bit of a bitmap is used to indicate whether the data block is free (0) or in-use (1)

https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

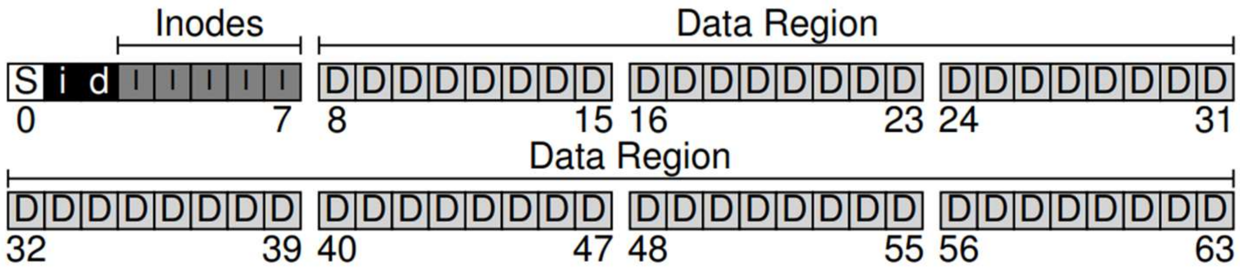
File system organization



- **Superblock**
  - Contains information about a file system
  - E.g. the number of inodes and data blocks in the file system
- When **mounting a file system**, the OS reads
  - The superblock first
  - Initialize various parameters
  - Attach the volume to the file-system tree
  - When files within the volume are accessed, the system will know exactly where to look for the needed on-disk structures

https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

File organization: Inode



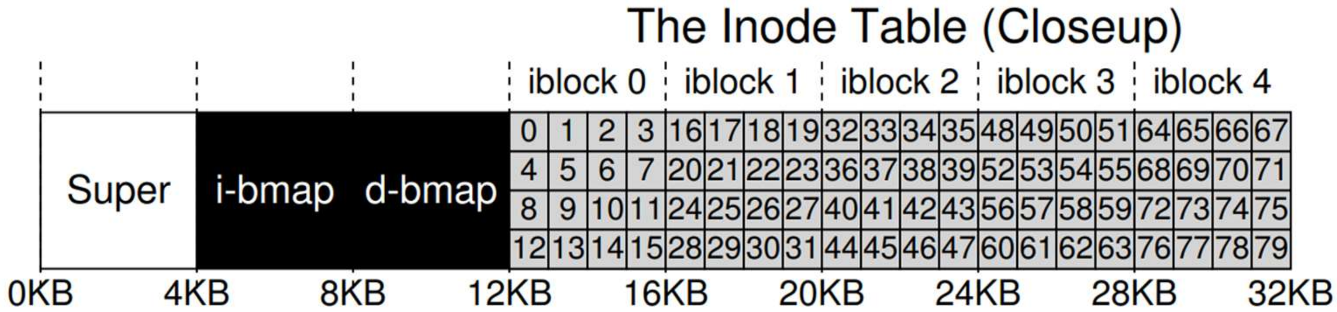
- **Inode (index node)**
  - Holds the **metadata** for a given file
  - **Contains all of the information that is needed about a file**
  - The length, permissions of a file, and the location of a file's block
- I-number
  - Used to calculate where on the disk the corresponding inode is located
  - E.g. the inode table as above takes 20 KB (five 4KB block)

https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

A file’s metadata (inodes)

- **Name**
  - The only information kept in human readable form
- **Identifier (inode number)**
  - A number that uniquely identifies the file within the file system
- **Type**
  - File type (inode based file, pipe, etc.)
- **Location**
  - Pointer to location of file on device
- **Size**
- **Protection**
  - Access control info. Owner, group (r, w, x) permissions, etc.
- **Monitoring**
  - Creation time, access time, etc.

File organization: inode

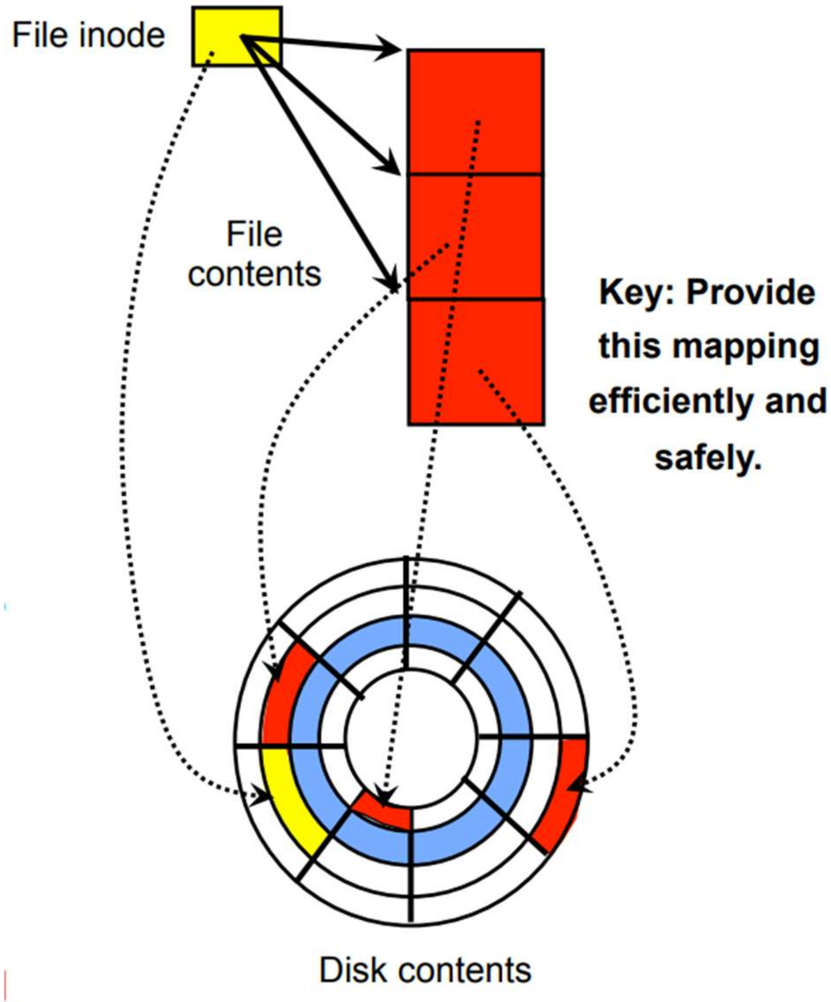


- **Read inode number 32**
  - Calculate the offset into the inode region
  - (32 * sizeof(inode)) = 8192
    sizeof(inode) = 256
  - Inode start at 12 KB (inodeStartAddr) in above case
  - Assuming a disk sector is 512 bytes, to fetch the block of inode 32
    - The file system issues a read to sector 20 x 1024 / 512 = 40
    - Blk = (inumber * sizeof (inode_t)) / blockSize;
    - Sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;

https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

File system data structures

- **Kernel (in-mem) structures**
  - Global open file table
  - Per-process open file table
  - Free (disk) block list
  - Free inode list
  - File buffer cache
  - Inode cache
  - Name cache
- **On-disk structures**
  - Superblock: file system format info
  - File: collection of blocks/bytes
  - File descriptor (inode): File metadata
  - Directory: Special kind of file
  - Free block/inode maps



https://my.eng.utah.edu/~cs5460/slides/Lecture17.pdf

Key in-memory data structures

- **Open file table**: shared by all processes with open file
  - Open count and “deleted” flag
  - Copy of (or pointer to) file’s inode
- **Per-process file table**: private for each process
  - Pointer to entry in global open file table
  - Current position in the file (“seek” pointer)
  - Access mode (read, write, read-write)
- **File buffer cache**: cache of file data blocks
  - Indexed by file-blocknum pairs (hash structure)
  - Used to reduce effective access time of disk operations

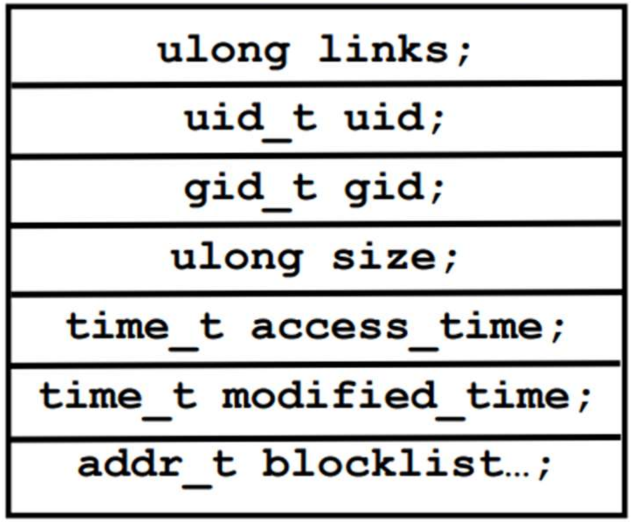
Key in-memory data structures

- **Name cache:** cache of recent name lookup results
  - Indexed by full filename (hash structure)
  - Used to decrease directory traversals for name lookups

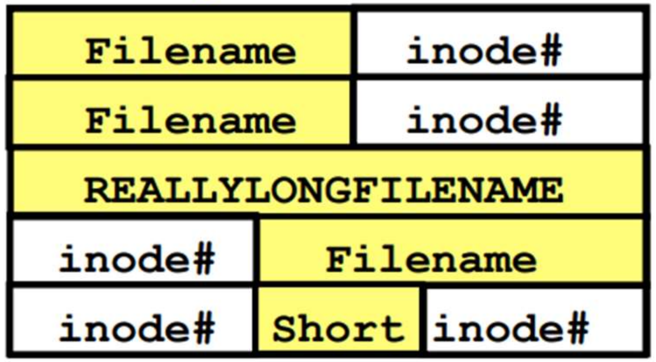
Key on-disk data structures

- **File descriptor (inode)**
  - Link count
  - Security attributes: UID, GID
  - Size
  - Access/modified times
  - “Pointers” to blocks
  - …
- **Directory file:**
  - File name (fixed/variable size)
  - Inode number
  - Length of directory entry
- **Free block/inode bitmap**
- **Superblock**

File descriptor (inode):



Directory file:



https://my.eng.utah.edu/~cs5460/slides/Lecture17.pdf

Buffer/page cache

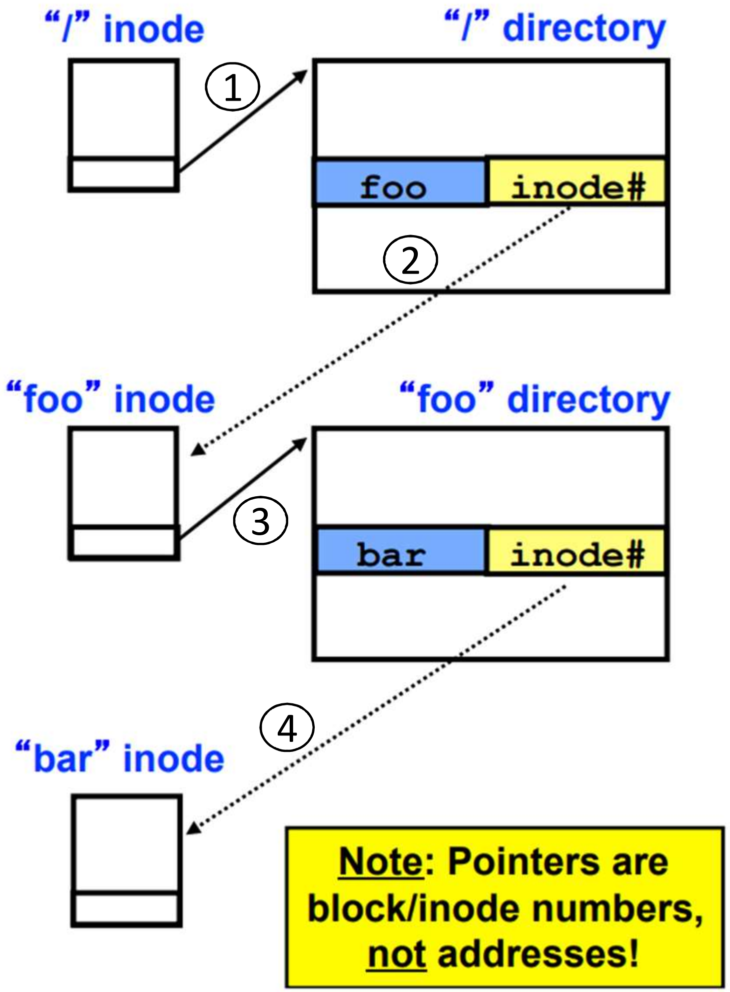
- **Idea**
  - **Keep recently used disk blocks in kernel memory**
- **Process reads from a file**
  - If blocks are not in page cache
    - Allocate space in page cache
    - Initiate a disk read
    - Block the process until disk operations complete
  - Copy data from page cache to process memory
  - Finally, system call returns
  - Usually, a process does not see the page cache directly
  - mmap() maps page cache pages into process RAM

Buffer/page cache

- **Process writes to a file**
  - If blocks are not in the page cache
    - Allocate pages
    - Initiate disk read
    - Block process until disk operations complete
  - Copy written data from process RAM to page cache
- Default: writes create dirty pages in the cache, then the system call returns
  - Data gets written to device in the background

Finding a file's inode on disk

- **Locate inode** for /foo/bar
  - 1. Find inode for "/"
    - Always in known location
  - 2. Read "/" directory into memory
  - 3. Find "foo" entry
    - If no match, fail lookup
  - 4. Load "foo" inode from disk
  - 5. Check permissions
    - If no permission, fail lookup
  - 6. Load "foo" directory blocks
  - 7. Find "bar" entry
  - 8. Load "bar" inode from disk
  - 9. Check permissions



https://my.eng.utah.edu/~cs5460/slides/Lecture17.pdf

Finding a file's blocks on disk

- **Inode consists of a table**
  - One entry per block in file
  - Entry contains physical block address (e.g., platter 3, cylinder 1, sector 26)
  - To locate data at offset X, read block (X / block_size)
- **Wants for inode table ?**
  - Most files are small
  - Most of disk is contained few large files
  - Need to efficiently support both sequential and random access
  - Want simple inode lookup and management mechanisms

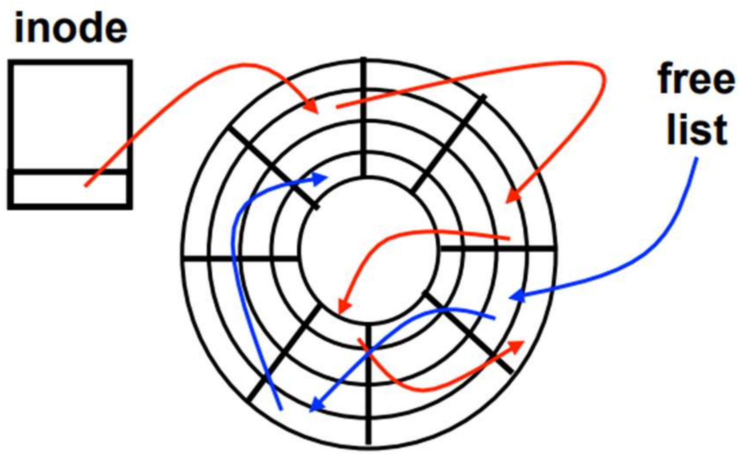
Allocating blocks to files

- **Contiguous allocation**
  - Files allocated (only) in contiguous blocks on disk
  - Analogous to base-and-bounds memory management
- **Linked file allocation**
  - Maintain a linked list of blocks used to contain file
  - At end of each block, add a (hidden) pointer to the next block
- **Indexed file allocation**
  - Maintain array of block numbers in inode
- **Multi-level indexed file allocation**
  - Maintain pointers to blocks full of more block numbers in inode

Contiguous allocation

- Files allocated in **contiguous blocks** on disk
- **Maintain ordered list of free blocks**
  - At create time, find large enough contiguous region to hold file
- Inode contains **START** and **SIZE**
- **Advantages**
  - Simple implementation
  - Easy offset ->block computation for sequential or random access
  - Few seeks
- **Disadvantages**
  - Fragmentation -> analogous to base and bounds
  - How do we handle file growth/shrinkage ?

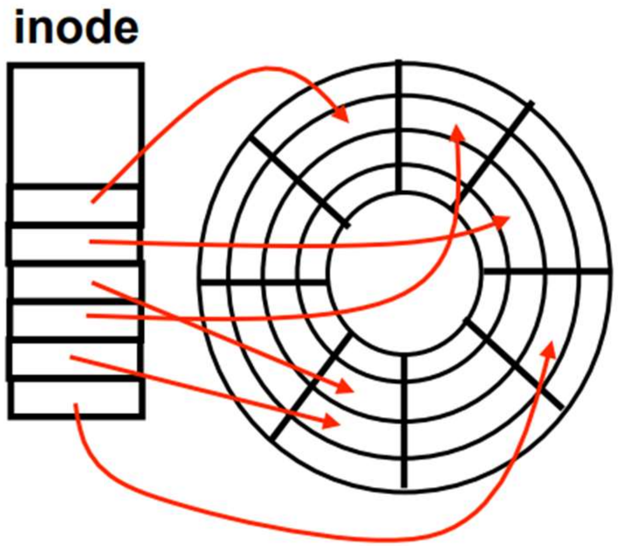
Linked file allocation



- **Linked list of free blocks**
  - Allocate any free blocks
- At end of each block, reserve space for block #
- Inode contains START
- **Good points**
  - Can extend/shrink files easily -> no fragmentation
  - Handles sequential accesses somewhat efficiently
- **Bad points**
  - Random access of large files is really inefficient
  - Lots of seeks -> non-contiguous blocks

https://my.eng.utah.edu/~cs5460/slides/Lecture18.pdf

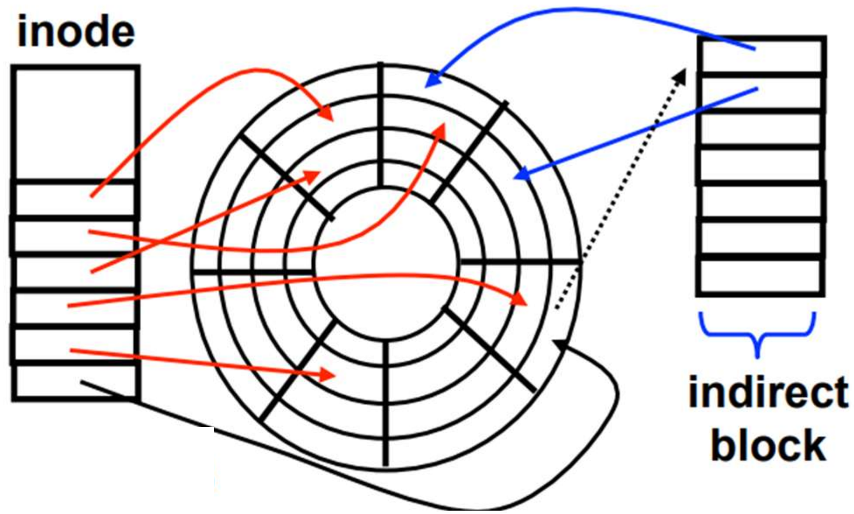
Indexed file allocation



- **Inode contains array of block addresses**
  - Allocate table at file creation time
  - File entries as blocks allocated
- Separate free block bitmap
- **Good points**
  - Can extend/shrink files to a point
  - Simple offset->block computation for sequential or random access
- **Bad points**
  - Variable sized inode structures
  - Lots of seeks-> non-contiguous blocks

https://my.eng.utah.edu/~cs5460/slides/Lecture17.pdf

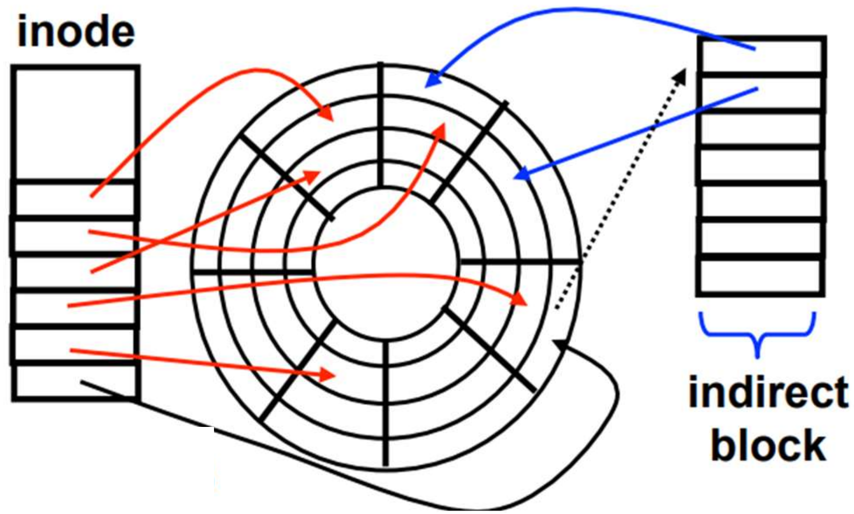
Multi-level indexed file allocation



- **Inode includes**
  - Fixed-size array of direct blocks
  - Small array of indirect blocks
  - Double/triple indirect (optional)
- **Indirection**
  - **Indirect pointer:** points to a block that contains more pointers
  - **Indirect block**: block full of block addresses
  - **Double indirect** block: block full of indirect block addresses
- **Use case: ext3**

https://my.eng.utah.edu/~cs5460/slides/Lecture17.pdf

Multi-level indexed file allocation



- **Good points**
  - Simple offset->block computation for sequential or random access
  - Allow incremental growth/shrinkage
  - Fixed size (small) inodes
  - Very fast access to (common) small files
- **Bad points**
  - Indirection adds overhead to random access to large files
  - Blocks can be spread all over disk -> more seeks

https://my.eng.utah.edu/~cs5460/slides/Lecture17.pdf

Multi-level indexed file allocation

- **Example: 4.3 BSD file system**
  - Inode contains 12 direct block addresses
  - Inode contains 1 indirect block address
  - Inode contains 1 double-indirect block address
- **How to support ever larger files ?**
  - Adds another pointer to the inode (double/triple indirect blocks)
- **If block addresses are 4-bytes and blocks are 2048-bytes, what is maximum file size in this file system ?**

Multi-level indexed file allocation

- **If block addresses are 4-bytes and blocks are 2048-bytes, what is maximum file size in this file system ?**
  - Number of block address per block = 2048 / 4 = 512
  - Number of blocks mapped by direct blocks = 12 (4.3 BSD file system)
  - Number of blocks mapped by indirect block = 512
  - Number of blocks mapped by double-indirect block = 512^2 = 262144
  - Max file size = (12 + 512 + 262144) * 2048 = ~ 513 MB (537,944,064 bytes)

Extents

- **An extent** is simply a disk pointer plus a length (in blocks)
  - (starting block, length)
  - A length to specify the on-disk location of a file
- **Each file is represented by a list of extents**
- **Pointer-based vs. extent-based**
  - Pointer-based is flexible but uses a large amount of metadata per file
  - Extent-based is less flexible but more compact
  - **Extent-based work well when there is enough free space on the disk and files can be laid out contiguously**
- **Use case: ext4**

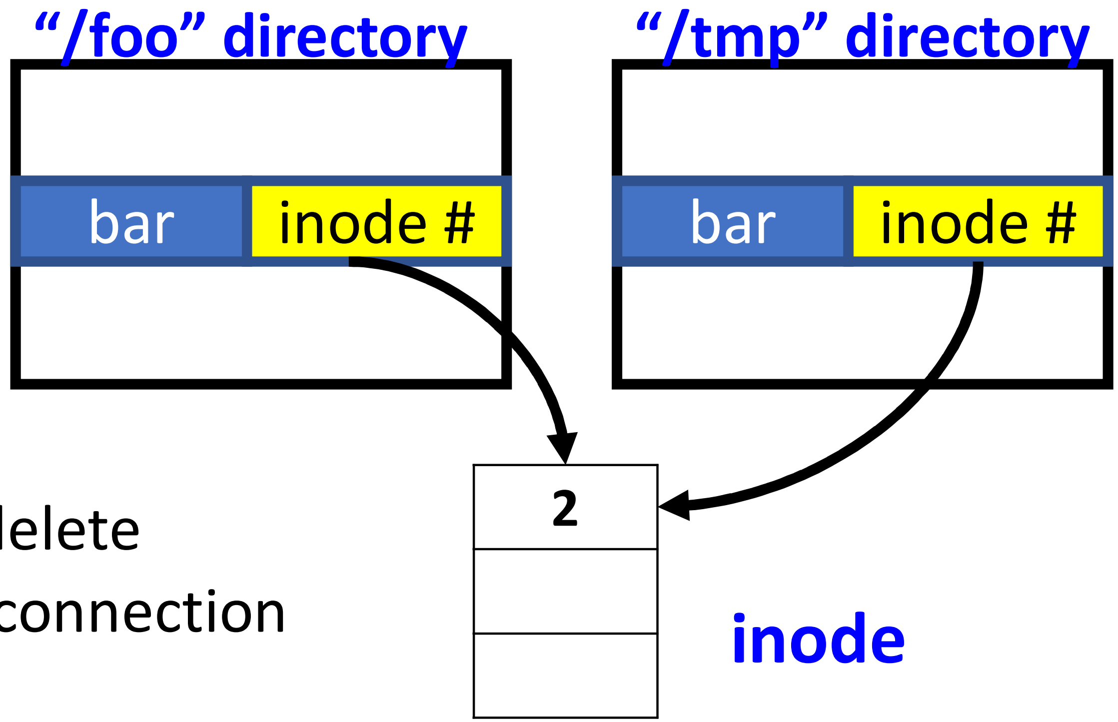
Linking

- **Links let us have multiple names to the same file**
- An inode uniquely identifies a file for its lifespan
  - Does not change when renamed
- Model: inode tracks “links” or references on disk
  - Count “1” for every reference on disk
  - Created by file names in a directory that point to the inode
- When link count is zero, inode (and contents) deleted
  - There is no ‘delete’ system call, only **‘unlink’**

Hard links

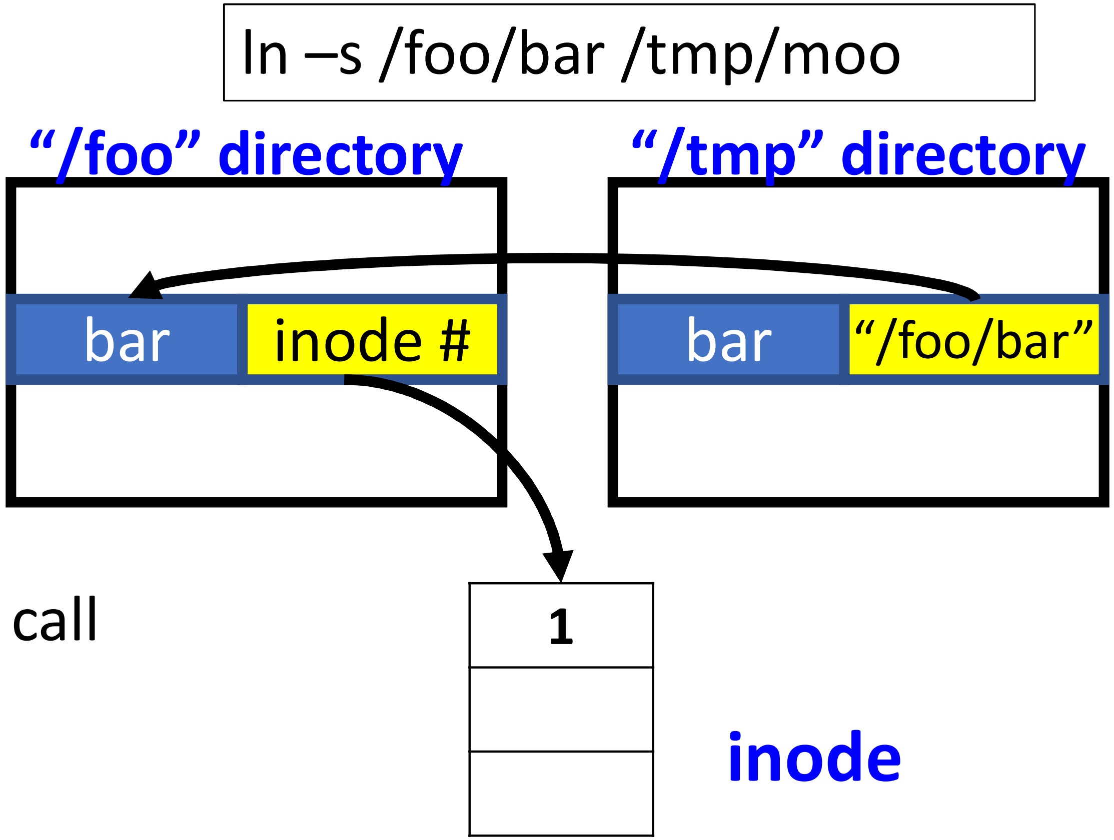
ln /foo/bar /tmp/moo

- **Hard links**
  - Two entries point to the same inode
  - Link count tracks connection
  - Decrement link count on delete
  - Only delete file when last connection is deleted
  - **Problem**: cannot cross file systems, unreachable directories



Soft links

- **Soft links**
  - Adds symbolic “pointer” to file
  - Special flag in directory entry
  - Created with symlink () system call
  - Only one “real” link to file
    - File goes away when its deleted



File allocation table (FAT) file system

- FAT file system
  - There are no inodes
  - Directory entries which store metadata about a file
  - Refer directly to the first block of said file
  - Impossible to create hard links

Mounting a file system

- Locate superblock(s)
- Read file system format information
- Initialize inode cache
- Initialize buffer cache
- Initialize name cache
- Optional: perform sanity checks
  - UNIX/ Linux / Mac OS X: fsck

Open (‘/foo/bar’) Operation

- **Open (“/foo/bar”, O_RDONLY)**
  - The file system first needs to **find the inode** for the file bar
  - Obtain the full pathname, than **traverse the pathname**
  - **All traversals begin at** the root of the file system (**root directory** ‘/’)
  - The FS **reads the inode of the root directory** based on i-number
  - **The root** has no parent, and **its inode number is 2** in UNIX
  - The FS **finds an entry for ‘foo’** from root’s inode
  - The FS **reads the block** including the inode of foo and its dir data
  - **Finds the inode number of bar**
  - **Read bar’s inode** into memory

Open (‘/foo/bar’) Operation

- **Open (“/foo/bar”, O_RDONLY)**
  - Once open, the problem can **issue a read ()** to read from the file
  - The first read will read the first block of the file
  - Consulting the inode to find the location of such a block
  - Update the inode with a new last-access time
  - Update and in-memory open file table for this file descriptor
- **In a open()**
  - Reading each block requires the file system to
    - first consult the inode
    - Read the block
    - Update the inode’s last-accessed-time

Write a file to disk

- **Write ()**
  - Writing to the file may also allocate a block unless the block is being overwritten
  - Need to write data to disk and decide which block to allocate to the file
- **Each write to a file logically generates 5 I/Os**
  - 1. **read the data bitmap** (mark the newly-allocated block as used)
  - 2. **write the bitmap** (reflect its new state to disk)
  - 3. **read and write the inode** (update with the new block's location)
  - 4. **write the actual block itself**

File creation

- **To create a file**
  - **Allocate** an inode
  - **Allocate** space within the directory containing the new file
  - One **read** to the inode bitmap (find a free inode)
  - One **write** to the inode bitmap (make it allocated)
  - One **write** to the new inode itself (initialize it)
  - One **write** to the data of directory (link high-level name of file to its inode number)
  - One **read and write** to the directory inode to update it
  - Additional I/Os if the directory needs to grow to accommodate the new entry (to the data bitmap and the new directory block)

Summary

- **File system organization**
  - **Blocks, inode, bitmap, superblocks**
- **File system data structures**
  - Open file table, file buffer cache, file descriptor etc.
- **Allocating blocks to the file**
  - Contiguous, linked, index, multi-level indexed file allocation, extent
- **Soft vs. hard link**