Lecture12 File System

1. Introduction

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.

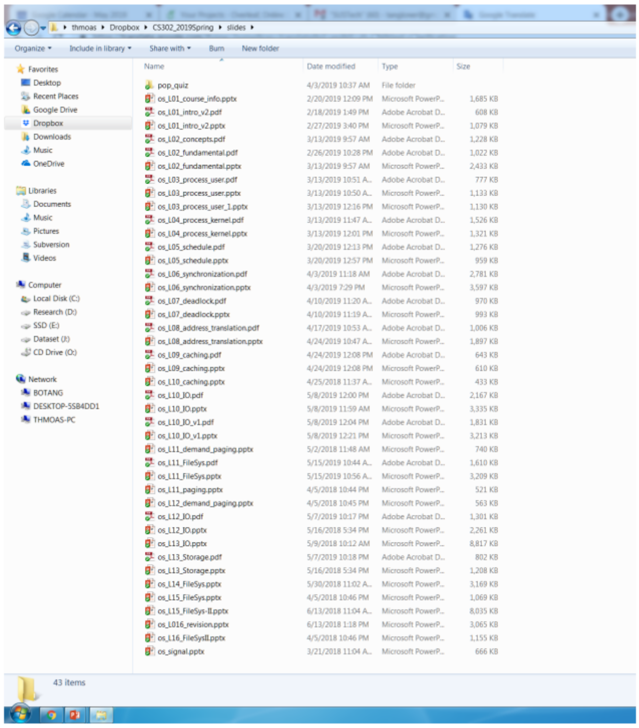
User vs. System View of a File

User's view	System's view (system call interface)	System's view (inside OS)
Durable data structures	Collection of **bytes** Doesn't matter to system what kind of data structures you want to store on disk!	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

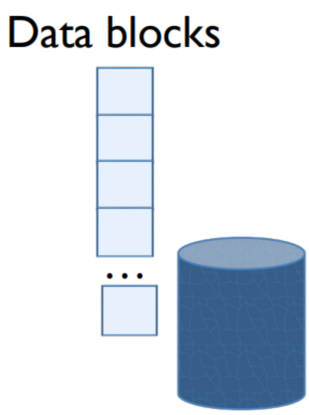
- What happens if user says: give me bytes 2—12?
    - Fetch block corresponding to those bytes
    - Return just the correct portion of the block
- What about: write bytes 2—12?
    - Fetch block
    - Modify portion
    - Write out Block
- Everything inside File System is in whole size **blocks**
- 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

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

File



- Named permanent storage
- Contains

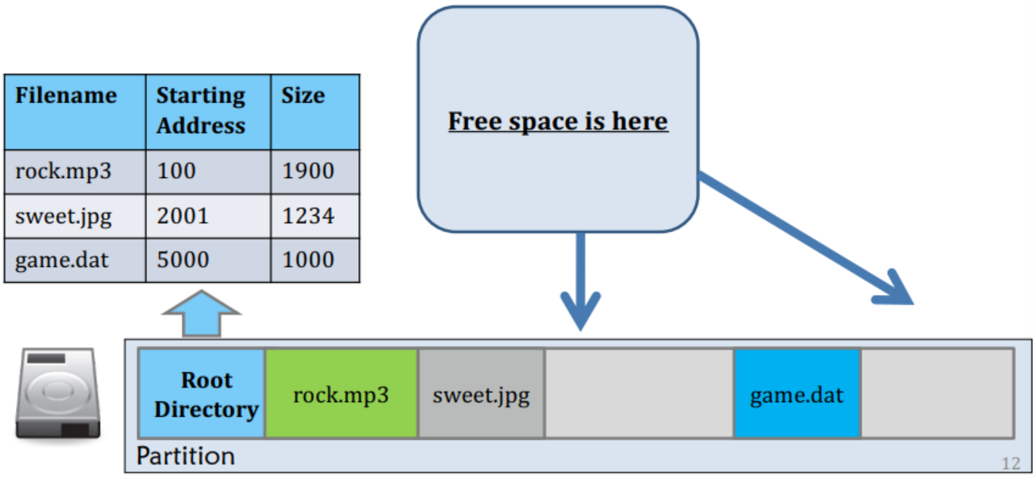
- Data
  - Blocks on disk somewhere
- Metadata (Attributes)
  - Owner size, last opened, ...
  - Access rights
    - R, W, X
    - Owner, Group, Other (in Unix systems)
    - Access control list in Windows system

Disk Management Policies

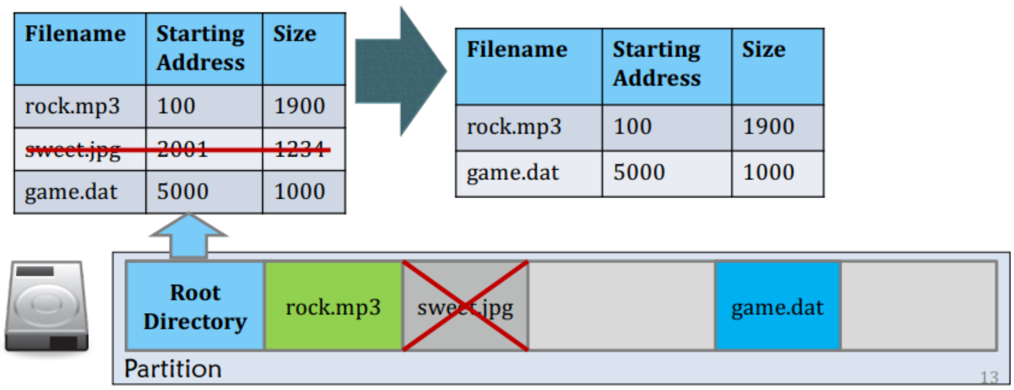
- Basic entities on a disk:
  - **File**: user-visible group of blocks arranged sequentially in logical space
  - **Directory**: user-visible index mapping names to files
- Access disk as linear array of sectors
  - Two Options:
    - Identify sectors as vectors [cylinder, surface, sector], sort in cylinder-major order, not used anymore
    - **Logical Block Addressing (LBA)**: 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
- 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

2. Contiguous Allocation

Locate files

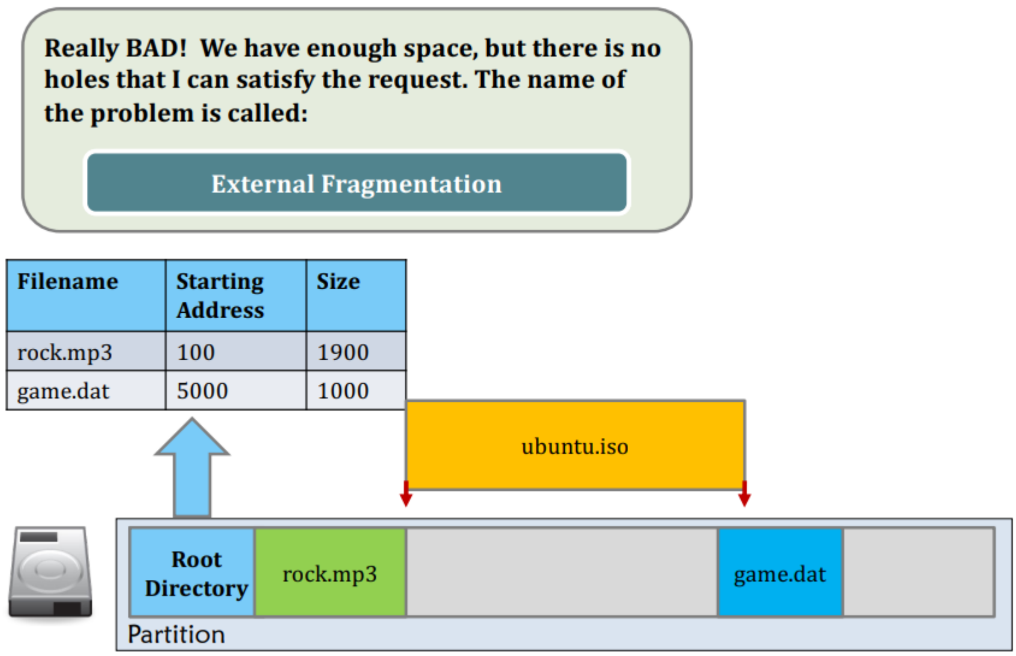


Delete files



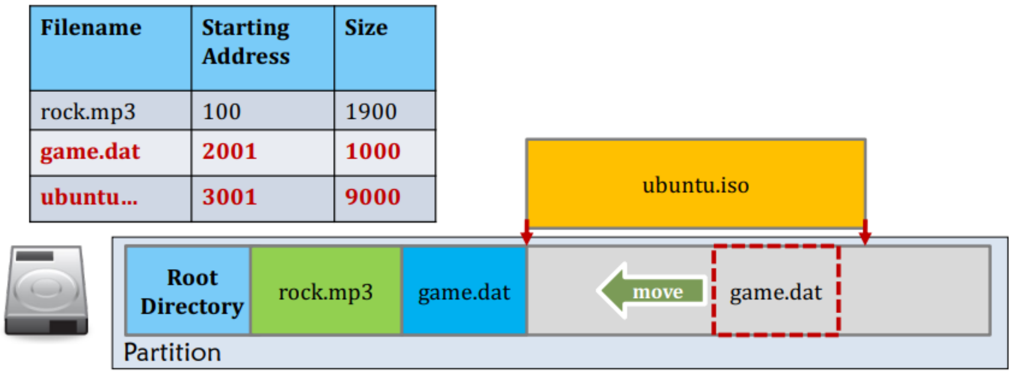
- Space de-allocation is the same as updating the root directory!

External Fragmentation



- Really BAD! We have enough space, but there is no holes that I can satisfy the request. The name of the problem is called: **External Fragmentation**

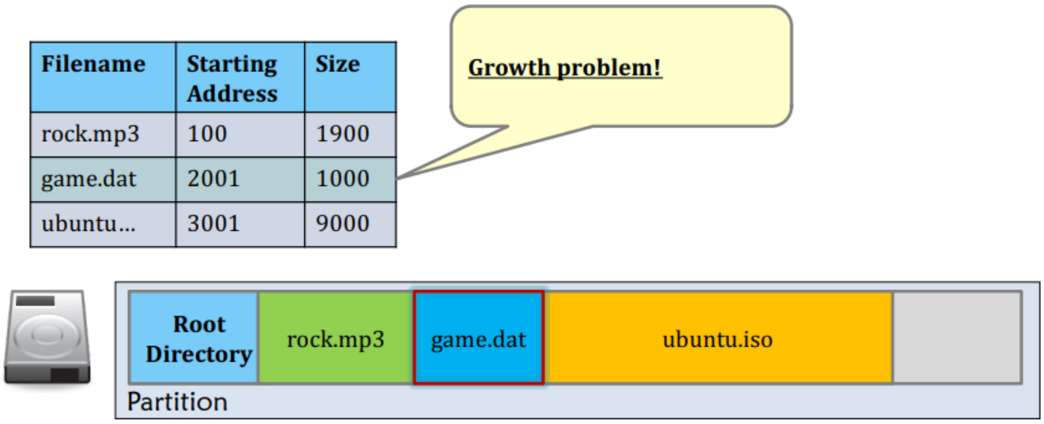
Defragmentation



Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	2001	1000
ubuntu...	3001	9000

- Defragmentation process may help!
- You know, this is very **expensive** as you're working on disks.

Growth Problem



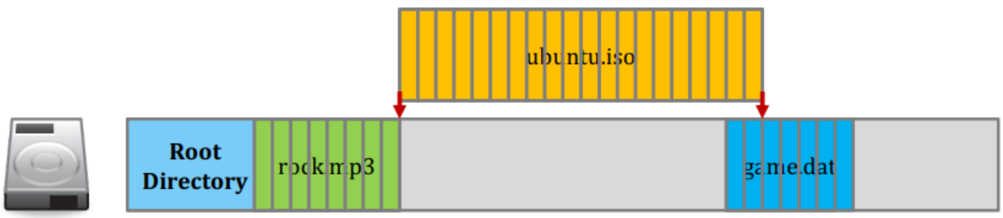
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	2001	1000
ubuntu...	3001	9000

Application

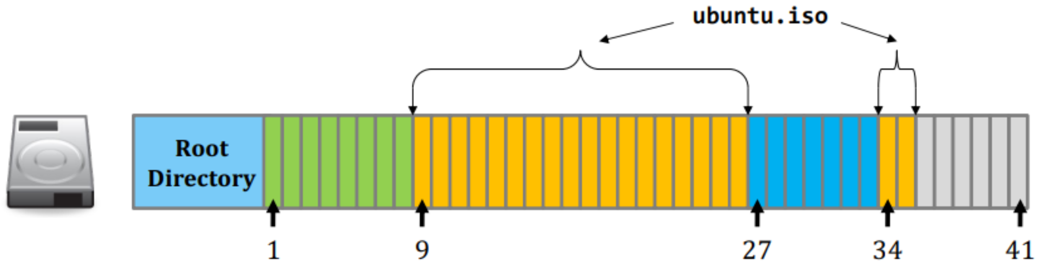
- ISO 9660
- CD-ROM
  - .iso image

3. Linked Allocation

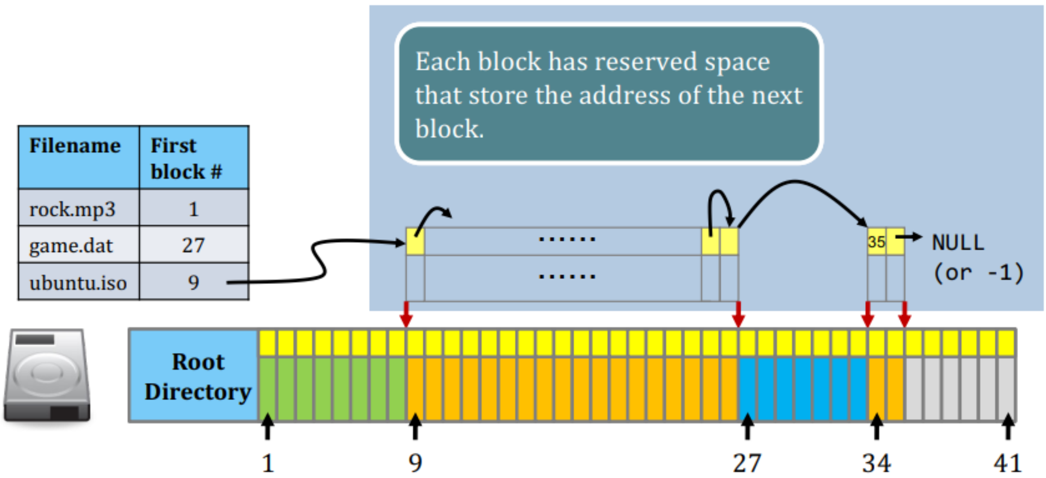
Locate files



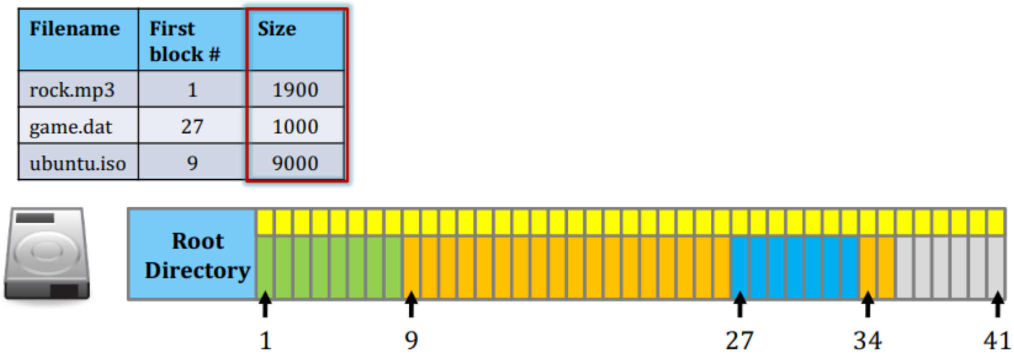
- Chop the storage device and data into **equal-sized blocks**



- Fill the empty space in a block-by-block manner



- Leave **4 bytes** from each block as the "**pointer**"
- To write the block # of the next block into the first 4 bytes of each block

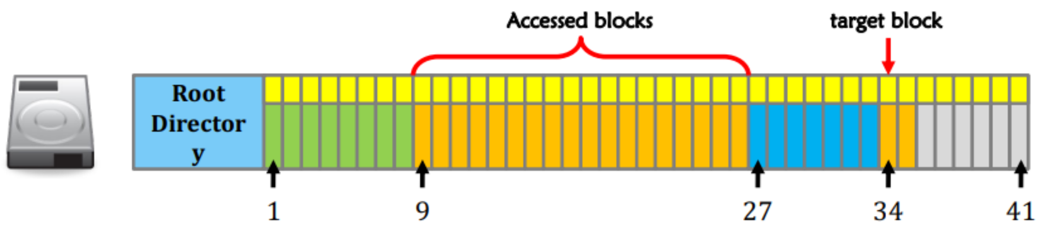


- Also keep the file size in the root directory table
- otherwise needs to live counting how many blocks each file has

Internal Fragmentation

- A file is not always a multiple of the block size.
    - The last block of a file may not be fully filled.
    - E.g., a file of size 1 byte still occupies one block.
- The remaining space will be wasted since no other files can be allowed to fill such space.

Poor random access performance



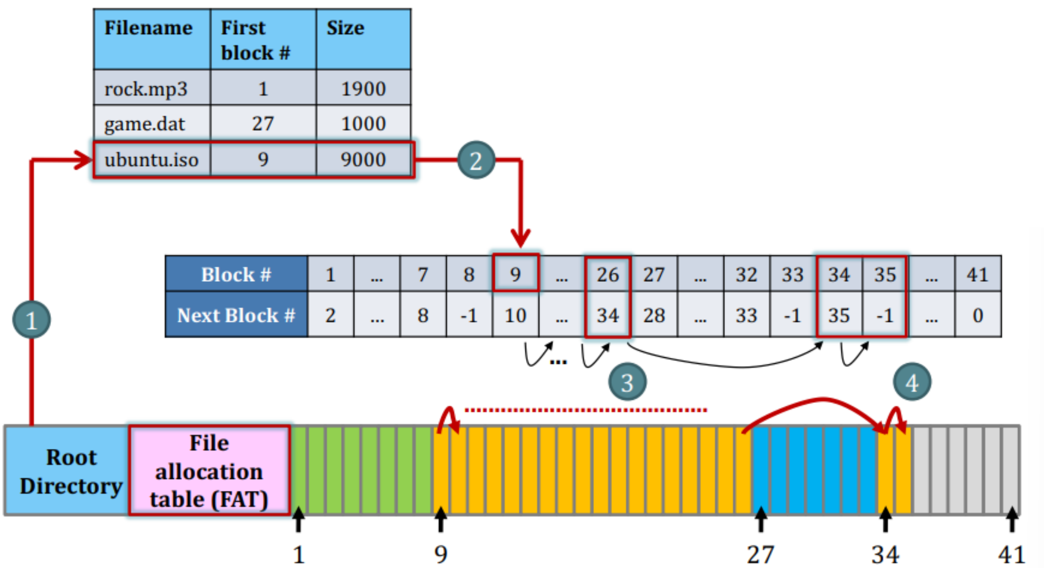
- What if I want to access the 2019-th block of ubuntu.iso?
- You have to access blocks 1 - 2018 of ubuntu.iso until the 2019-th block

4. FAT

- Centralize all the block links as File Allocation Table

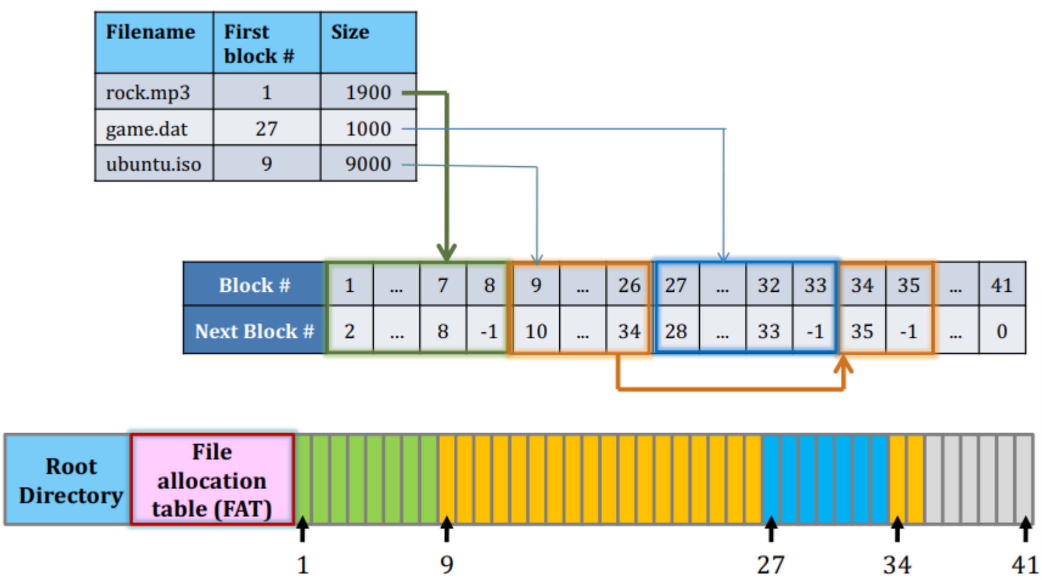
Example

Task: read `ubuntu.iso` sequentially



- Step1: Read the **root directory** and retrieve the **first block number**
- Step2: Read the **FAT** to determine the **location of the next block**

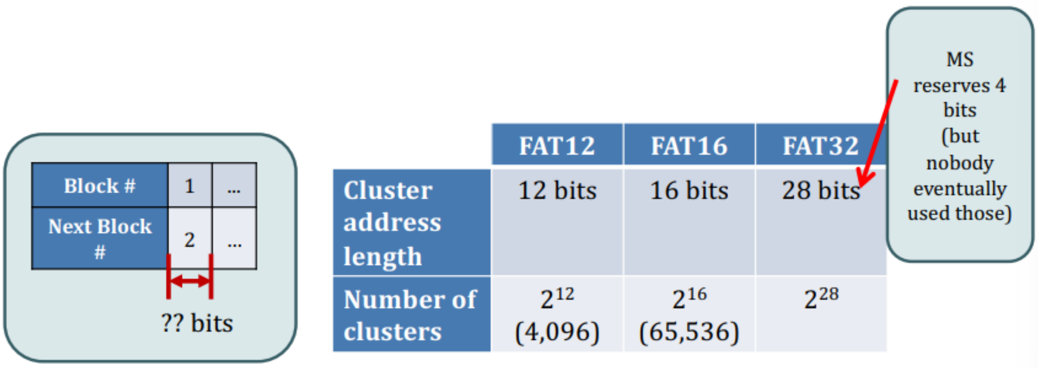
- Step3: After reading the 2nd block, the process continues. Note that the blocks **may not be contiguously allocated**
- Step4: The process stops until the FAT says the next block # is -1



- Resulting layout & file allocation

FAT Size

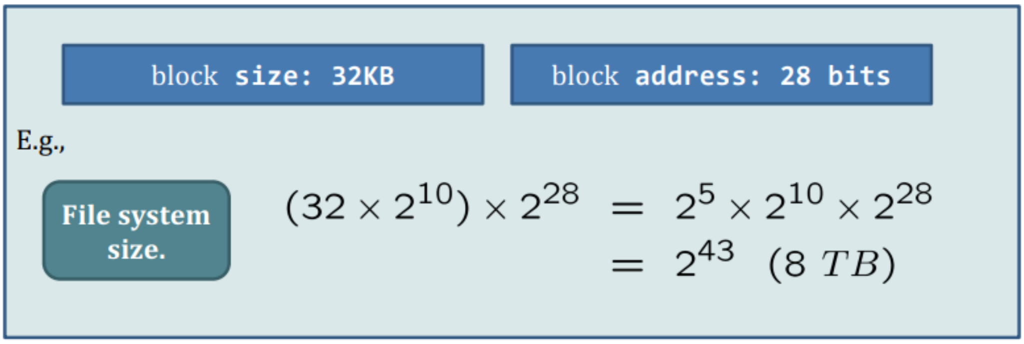
- Start from floppy disk and DOS
    - On DOS, a block is called as a "**cluster**"



- Size of FAT: FAT12, 12-bit cluster address, can point up to 2^{12} = 4096 blocks
- Size of block(cluster)

Available block sizes (bytes)								
512	1K	2K	8K	16K	32K	64K	128K	256K

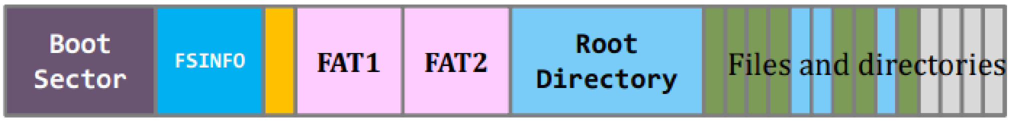
File system size:



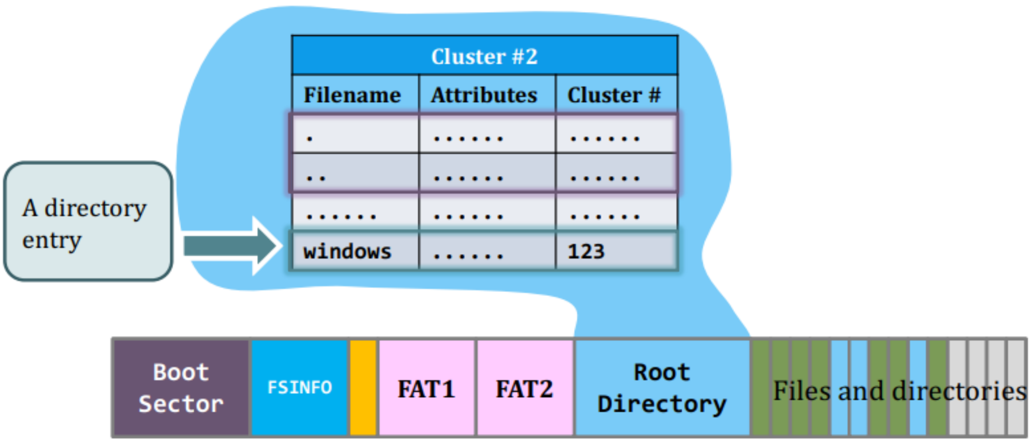
FAT series

Layout overview

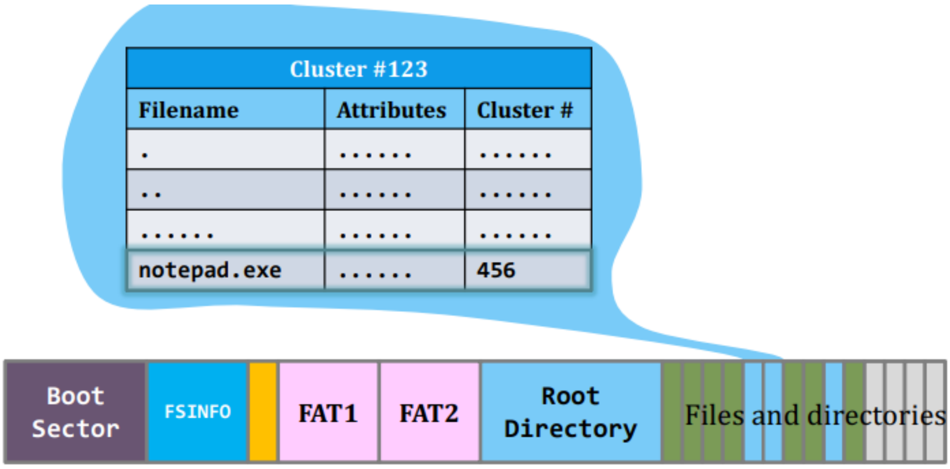
	Propose	Size
Boot sector	FS-specific parameters	1 sector, 512 bytes
FSINFO	Free-space management	1 sector, 512 bytes
More reserved sectors	Optional	Variable, can be changed during formatting
FAT (2 pieces)	1 copy as backup	Variable, depends on disk size and cluster size.
Root directory	Start of the directory tree.	At least one cluster, depend on the number of directory entries.



Directory traversal



- Step1: Read the directory file of the root directory starting from Cluster #2
  - `C:\windows` starts from Cluster #123



- Step2: Read the directory file of the `C:\windows` starting from Cluster #123

Directory entry

A 32-byte directory entry in a directory file

A directory entry is describing a **file** (or a sub-directory) under a **particular directory**

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	remaining characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

Filename	Attributes	Cluster #
explorer.dat	......	32

0	e	x	p	l	o	r	e	r	7
8	e	x	e	...	...	...	...	...	15
16	...	...	...	...	00	00	...	...	23
24	...	...	20	00	00	C4	0F	00	31

- This is a 8+3 naming convention
  - 8 characters for name
  - 3 characters for file extention

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

Filename	Attributes	Cluster #
explorer.dat	......	32

0	e	x	p	l	o	r	e	r	7
8	e	x	e	...	...	...	...	...	15
16	...	...	...	...	00	00	...	...	23
24	...	...	20	00	00	C4	0F	00	31

35

- The 1st block address of that file

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

Filename	Attributes	Cluster #
explorer.dat	......	32

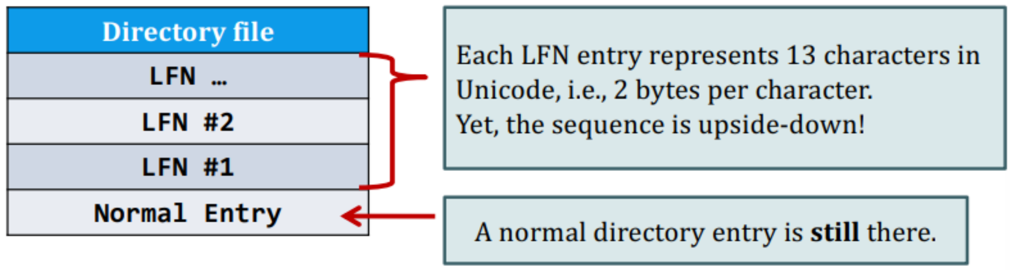
0	e	x	p	l	o	r	e	r	7
8	e	x	e	...	...	...	...	...	15
16	...	...	...	...	00	00	...	...	23
24	...	...	20	00	00	C4	0F	00	31

- The largest size of a FAT32 file: 4G - 1 bytes
  - We need to represent 0 bytes

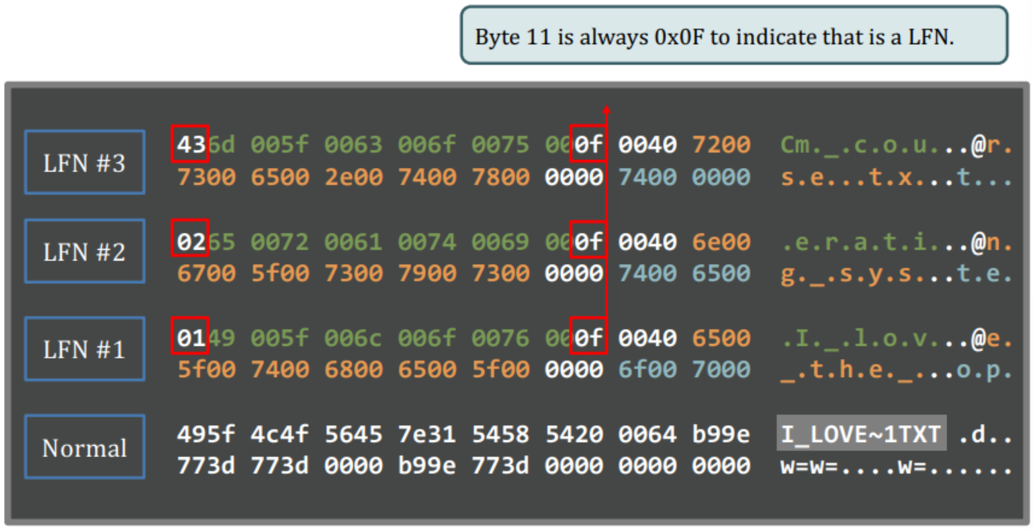
LFN Directory entry

LFN: Long File Name

- In old days, Uncle Bill set the rule that every file should follow the 8+3 naming convention.
- To support LFN
  - Abuse directory entries to store the file name
  - Allow to use up to 20 entries for one LFN



"I_love_the_operating_system_course.txt".



Normal directory entry v.s. LFN directory entry

Bytes	Description
0-0	1st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

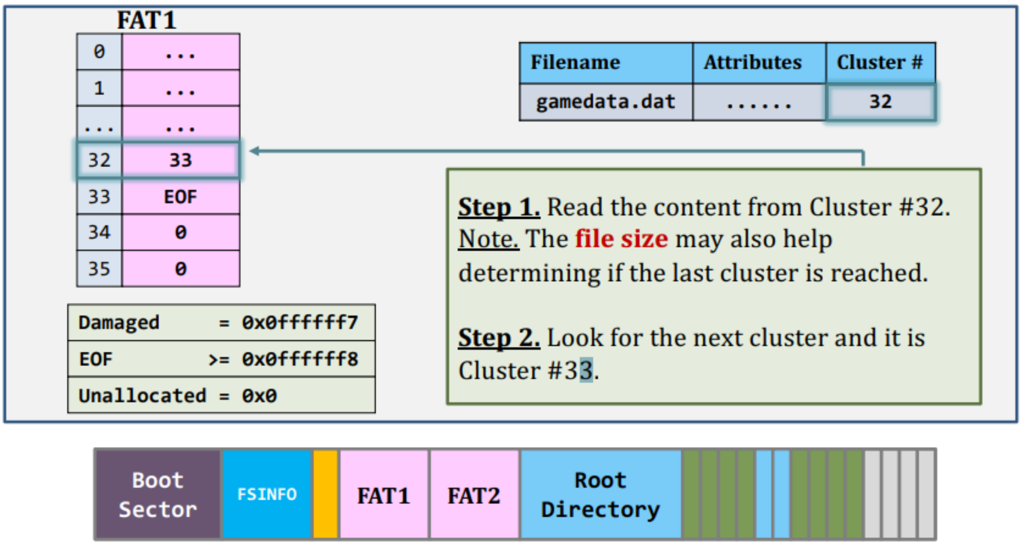
Bytes	Description
0-0	Sequence Number
1-10	File name characters (5 characters in Unicode)
11-11	File attributes - always 0x0F (to indicate it is a LFN)
12-12	Reserved.
13-13	Checksum
14-25	File name characters (6 characters in Unicode)
26-27	Reserved
28-31	File name characters (2 characters in Unicode)

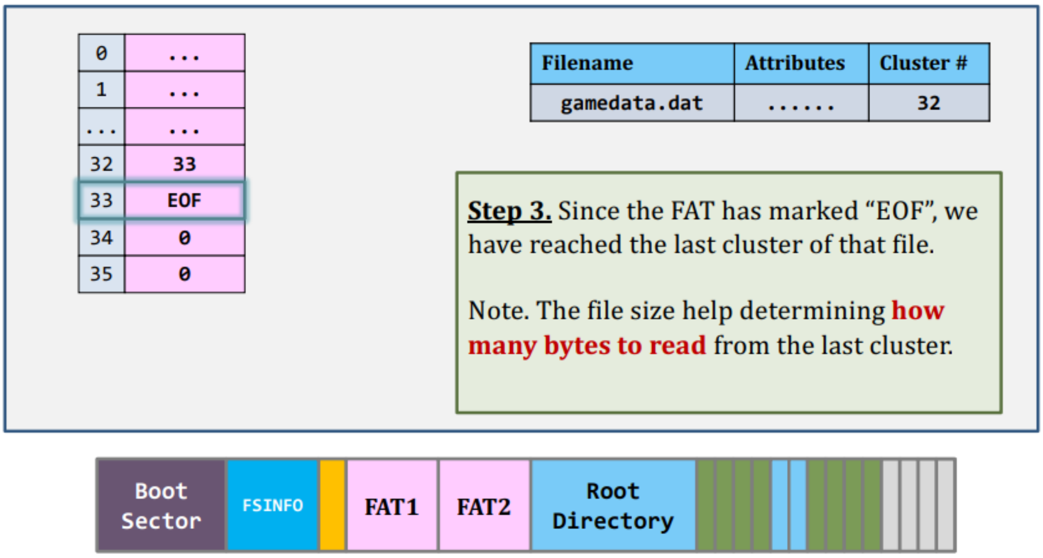
Summary

- A **directory** is an extremely important part of a FAT-like file system.
    - It stores the **start cluster number**.
    - It stores the **file size**; without the file size, how can you know when you should stop reading a cluster?
    - It stores **all file attributes.**

Reading a file

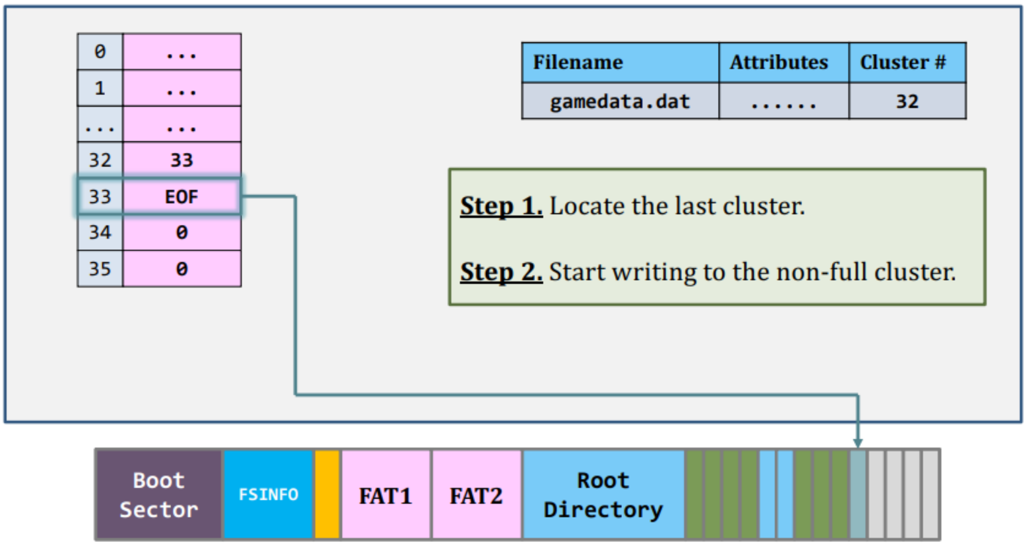
Task: read `C:\windows\gamedata.dat` sequentially

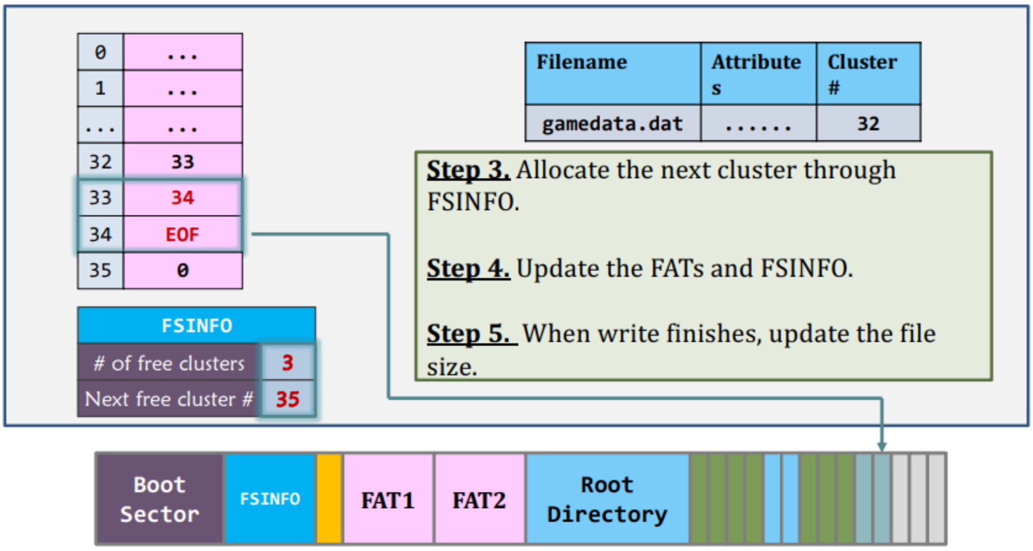


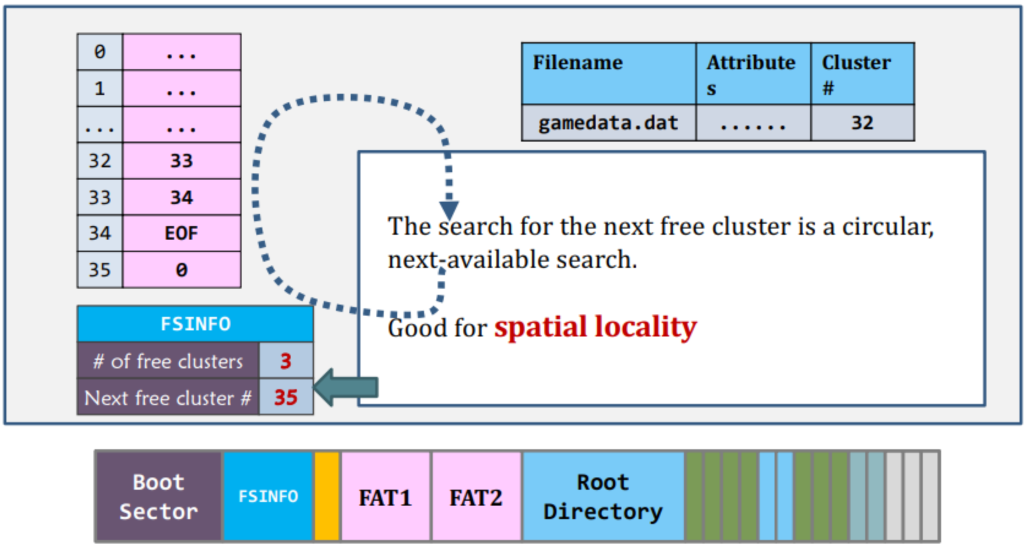


Writing a file

Task: append data to `C:\windows\gamedata.dat`

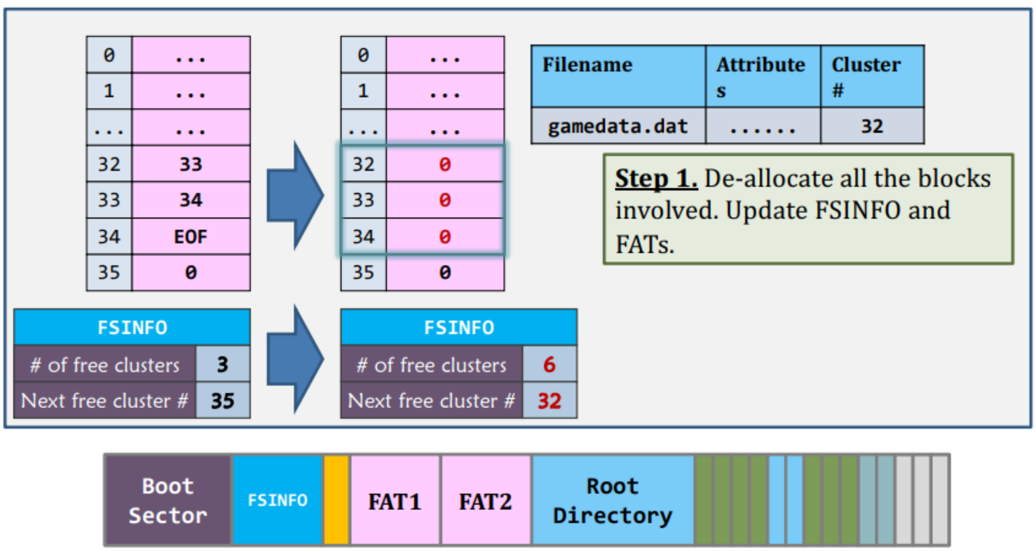


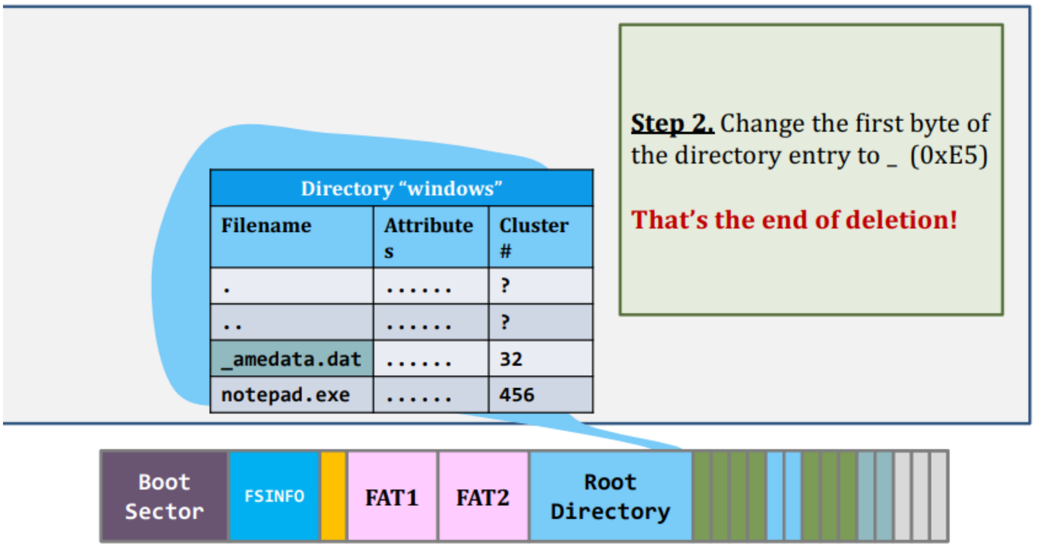




Delete a file

Task: delete `C:\windows\gamedata.dat`





The file is not really removed from the FS layout

- Perform a search in all the free space. Then, you will find all deleted file contents.

"Deleted data" persists until the de-allocated clusters **are reused**.

- This is an issue between performance (during deletion) and security.

Recover a deleted file

If you really care about the deleted file

- Pull the power plug at once
- Pulling the power plug stops the target clusters from being over-written

File size is within one block (cluster)	Because **the first cluster address** in the direct is still readable, the recovery is having a very high successful rate.
File size spans more than 1 block	Because of the next-available search, clusters of a file are likely to be contiguous allocated. This provides a hint in looking for deleted blocks. Can you devise an undelete algorithm for FAT32?

5. Inode Allocation

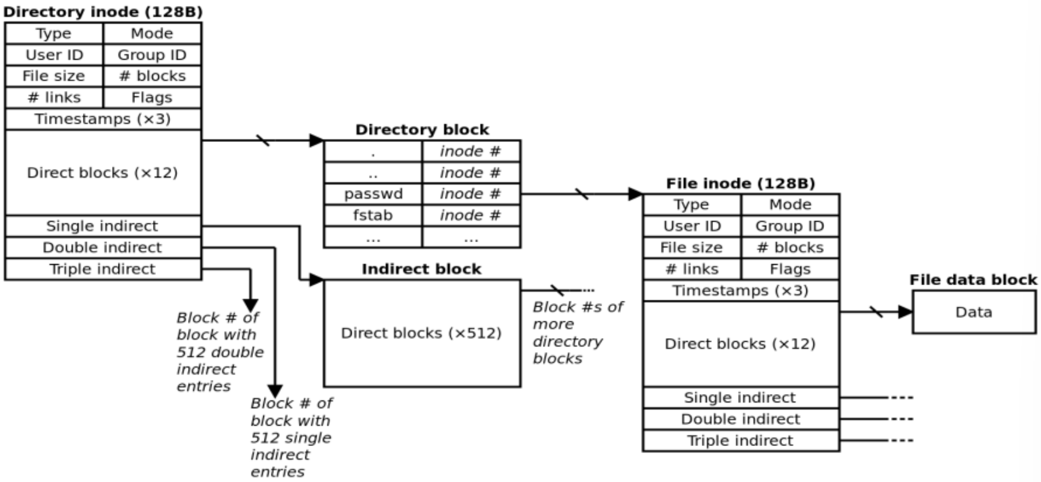
Unix File System

- Original iNode format appeared in BSD 4.1
    - Berkely Standard Distribution Unix
    - Similar structure for Linux Ext2/3
- File Number is index of iNode arrays
- Multi-level index structure

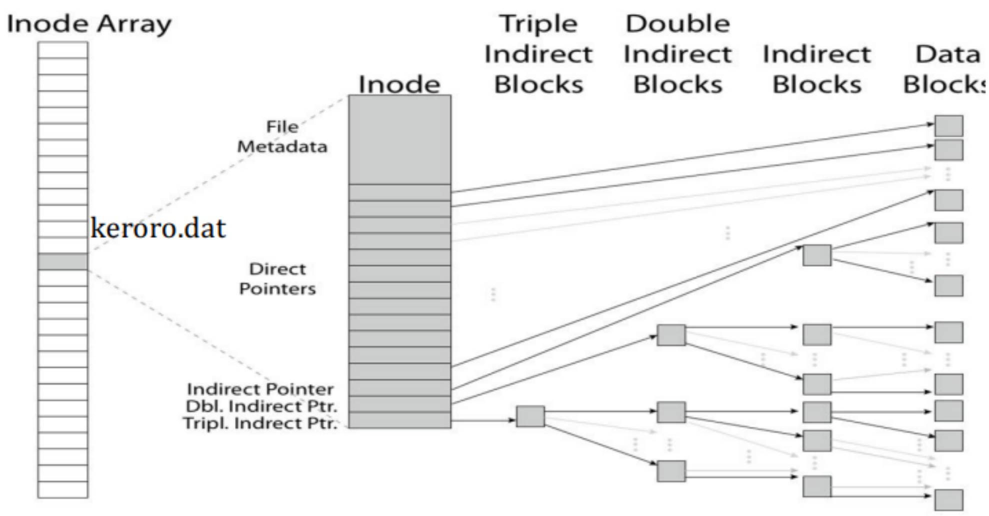
- Great for little and large files
- Unbalanced tree with fixed sized blocks
- Metadata associated with file
    - Rather than in the directory that points to it
- Scalable directory structure

iNode

One directory/file has one iNode



- iNode table is an array of iNodes
- Pointers are unbalanced tree-based data structures



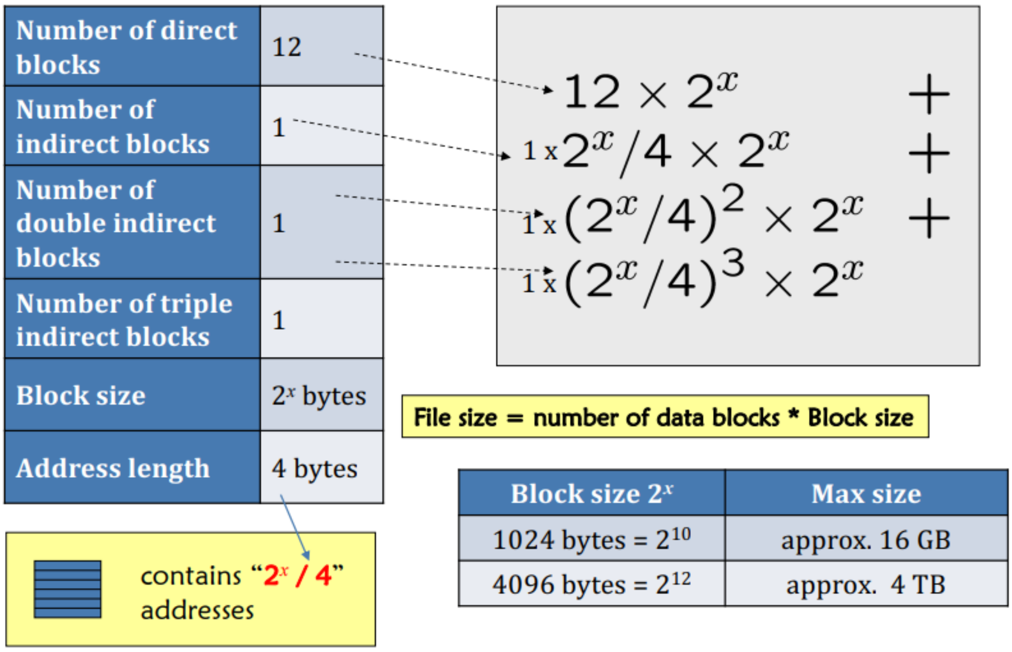
Structure

- iNode metadata
    - User
    - Group
    - 9 basic access control bits: UGO x RWX
- small files: 12 pointers direct to data block
    - data pointers:

- 4KB blocks -> sufficient for files up to 48KB
- indirect pointers
    - point to a disk block
        - containing only pointers
    - 4KB blocks -> 1024 ptrs
        - 4MB level2
        - 4GB level3
        - 4TB level4

File size

file size = number of data blocks * block size



= $12 \times 2^x + 2^{2x-2} + 2^{3x-4} + 2^{4x-6}$

6. File System Ext

The latest default FS for Linux distribution is the **Fourth Extended File System**, Ext4 for short.

For Ext2 & Ext3

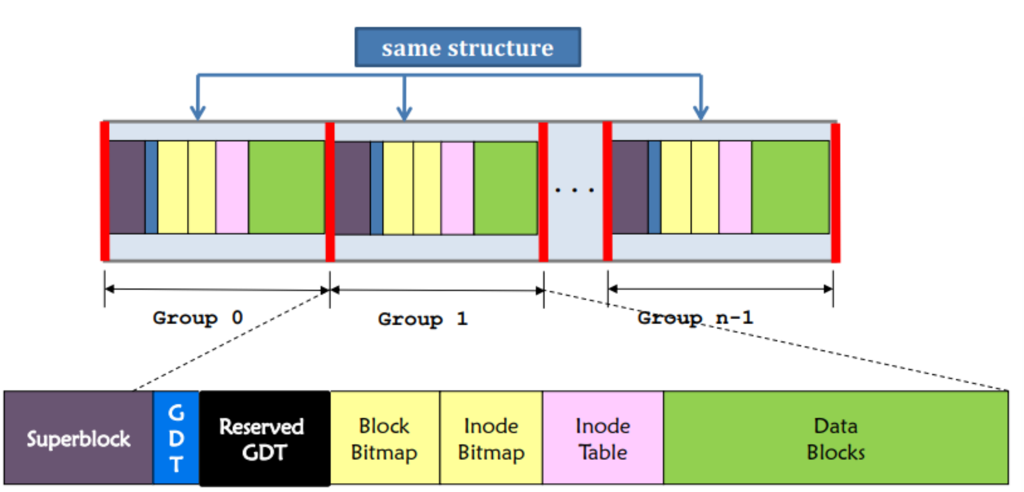
- Block size: 1024, 2048 or 4096 bytes
- Block address size: 4 bytes -> # of block addresses = 2^{32}

$2^x \times 2^{32} = 2^{32+x}$			
Block size	$2^x = 1024$	$2^x = 2048$	$2^x = 4096$
File System size	4 TB	8 TB	16 TB

Ext2/3 - Disk Layout

Block groups

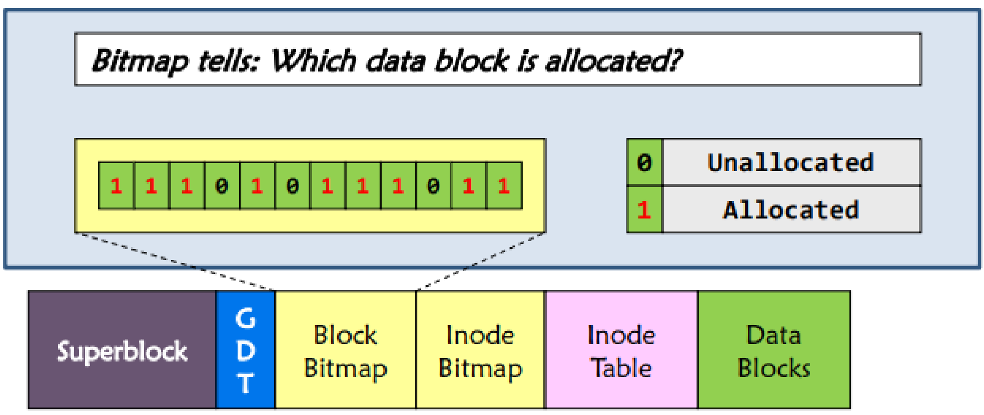
The file system is divided into block groups and every block group has the same structure



FS layout

Superblock	Stores FS specific data. E.g., the total number of blocks, etc.
GDT – Group Descriptor Table	It stores: - The locations of the **block bitmap**, the **iNode bitmap**, and the **iNode table**. - Free block count, free iNode count, etc...
Block Bitmap	A bit string that represents if a block is allocated or not.
iNode Bitmap	A bit string that represents if an inode (index-node) is allocated or not.
iNode Table	An array of inodes ordered by the inode #.
Data Blocks	An array of blocks that stored files.

Block Bitmap



iNode Bitmap

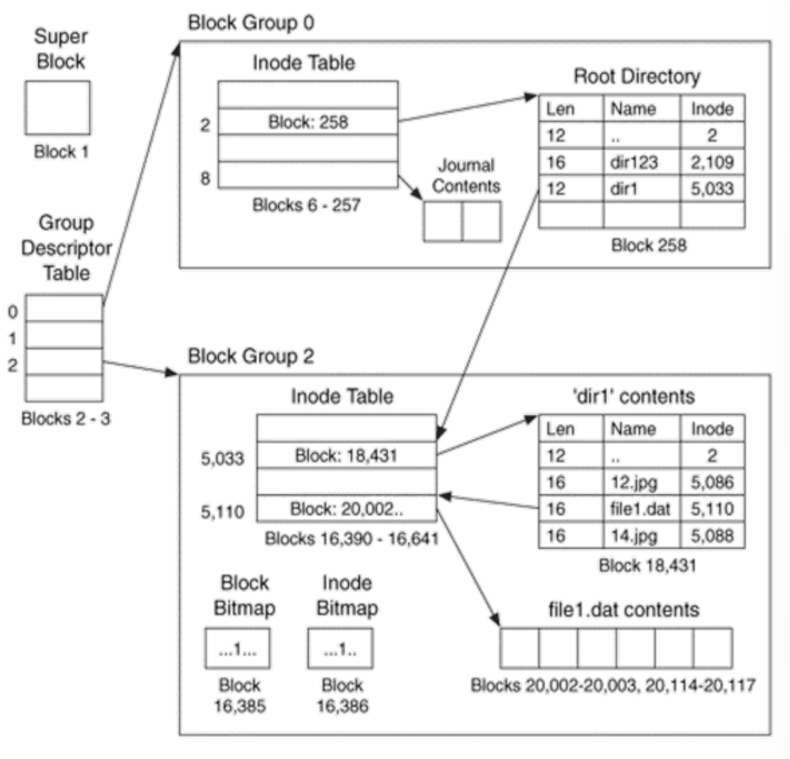
- A bit string that represents if an iNode (index-node) is allocated or not
  - implies that the number of files in the file system is fixed

Block groups

Why having groups

- **Performance**: spatial locality
  - Group iNodes and data blocks of related files together
- **Reliability**: superblock and GDT are replicated in each block group

Linux Example



- Disk divided into block groups
  - Each group has two block-sized bitmaps (free blocks/inodes)

Ext 2/3 Directory

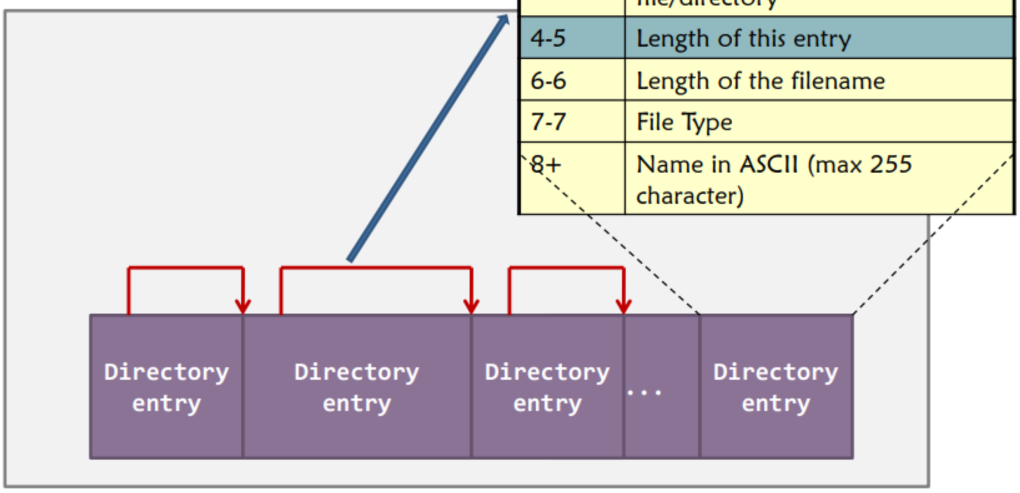
iNode structure

iNode Structure (128 bytes long)	
Bytes	**Value**
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count (will discuss later)
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
108-111	Upper 32 bits of file sizes in bytes

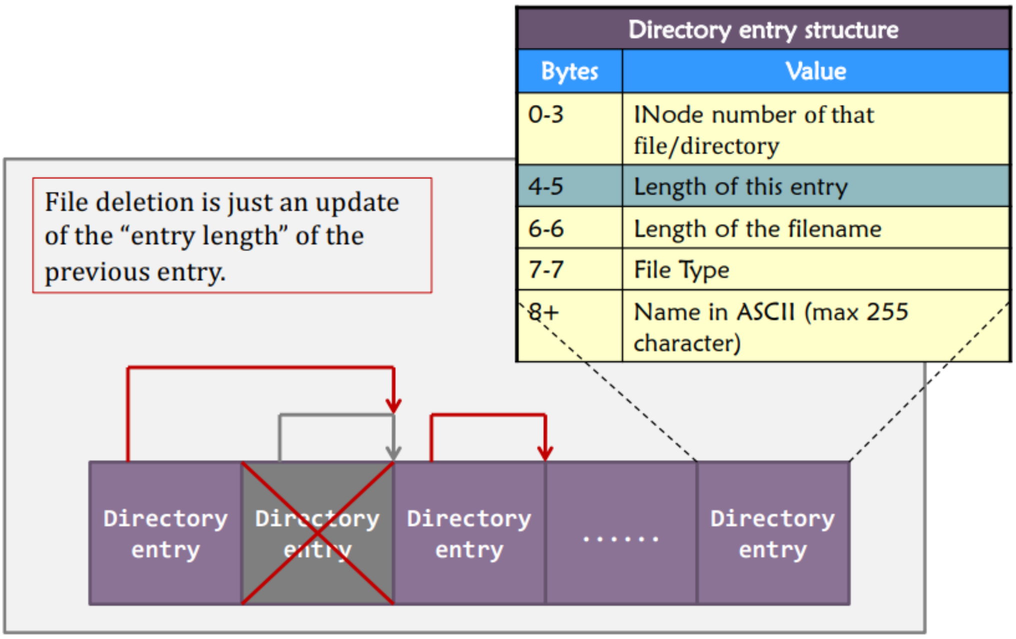
The locations of the data blocks are stored in the inode.

Directory entry in a directory block

Directory entry structure	
Bytes	**Value**
0-3	iNode number of that file/directory
4-5	Length of this entry
6-6	Length of the filename
7-7	File Type
8+	Name in ASCII (max 255 character)



File deletion



Ext 2/3 - Hard and Soft Links

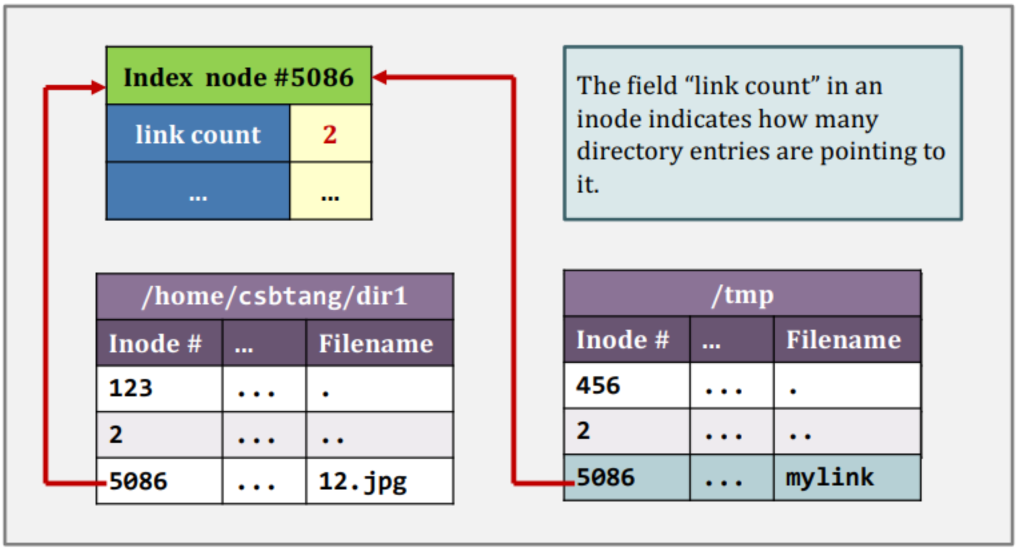
Hard link

A hard link is a directory entry pointing to the iNode of an existing file.

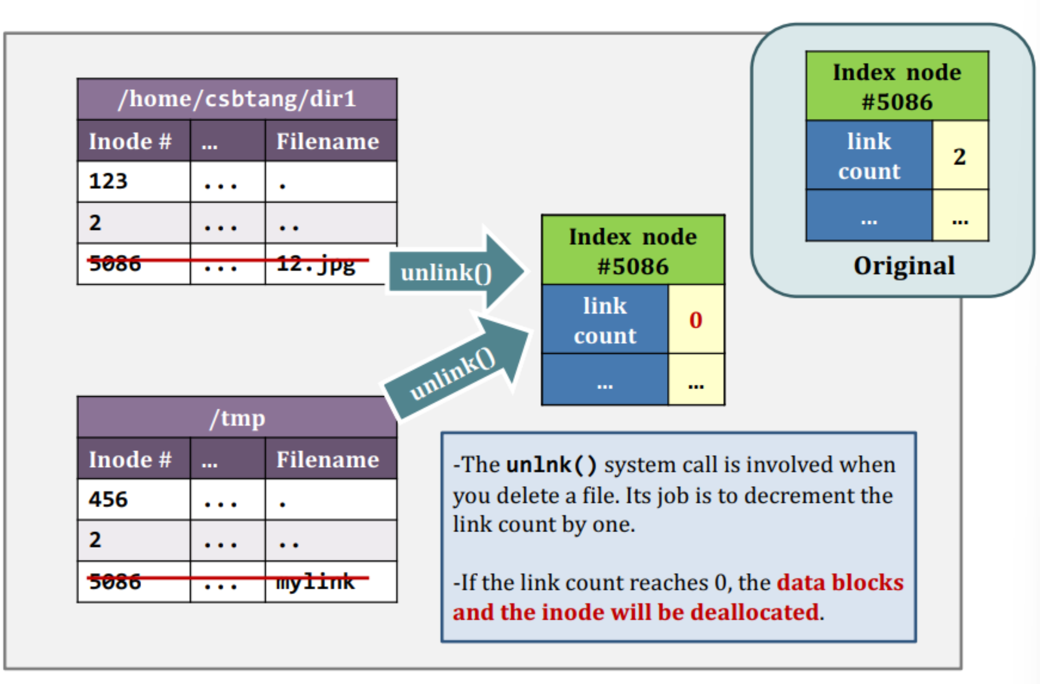
- **the file can be accessed through two different pathnames**

/home/csbtang/dir1		
Inode #	**...**	**Filename**
123	...	.
2	...	..
5086	...	12.jpg

/tmp		
Inode #	**...**	**Filename**
456	...	.
2	...	..
5086	...	mylink

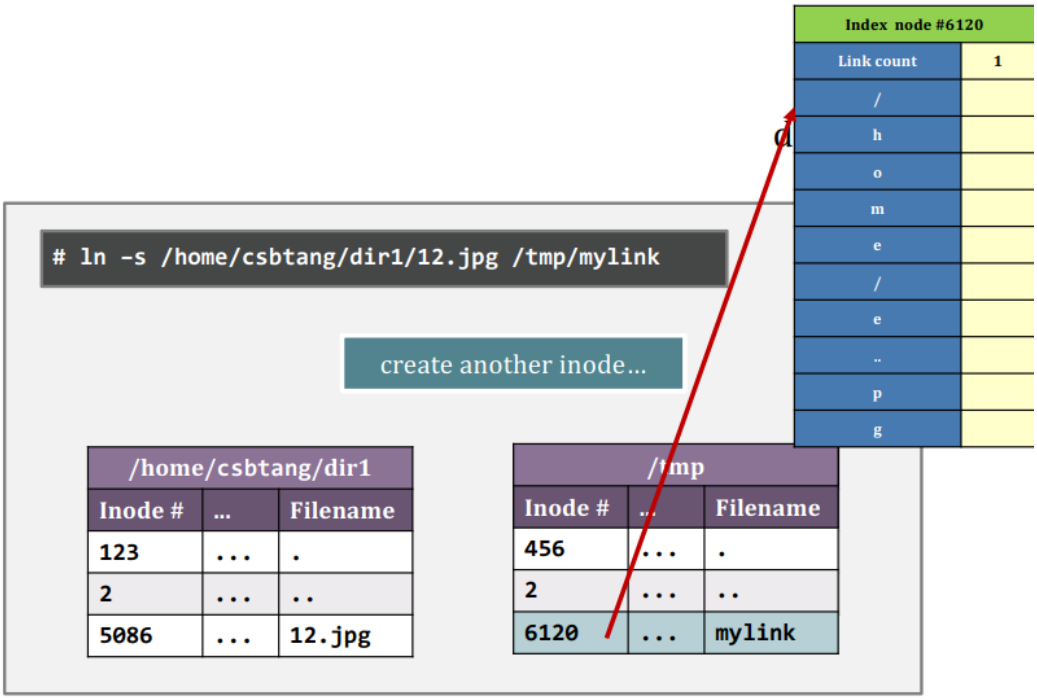


Removing file and link count



Soft link/ Symbolic link

A soft/symbolic link creates a new inode



- Symbolic link is pointing to a new iNode whose target's **pathname** are stored using the space originally designed for **12 direct block and the 3 indirect block pointers** if the pathname is shorter than **60 characters**.
  - Use back a normal inode + one direct data block to hold the longpathname otherwise

Summary of Links

Hard link

- Sets another directory entry to contain the file number for the file
- Creates another name (path) for the file

- Each is "first class

Soft link

- Directory entry contains the path and name of the file
- Map one name to another one