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.

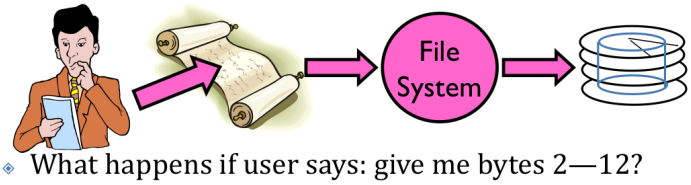
User vs. System View of a File

- 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

内存块的集合

内存块的大小 ≥ sector的大小.

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

修改是整个block一起修改.

Directory

层次架构

- Basically a hierarchical structure

- Each directory entry is a collection of
  - Files
  - Directories
    - A link to another entries

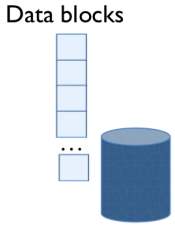
- Each has a name and attributes
  - Files have data

- Links (hard links) make it a DAG, not just a tree
  - Softlinks (aliases) are another name for an entry

硬连接是 DAG

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



元数据：一些基本属性

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

disk被映射成 sector 的线性数组.

- 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

文件头保存的数据能追踪文件所在的block.

File System Layout

- contiguous allocation
- linked allocation
- inode allocation (next lecture)

数组
链表
节点

Continuous allocation

Locate files easily.

Filename	Starting Address	Size
rock.mp3	100	1900
sweet.jpg	2001	1234
game.dat	5000	1000

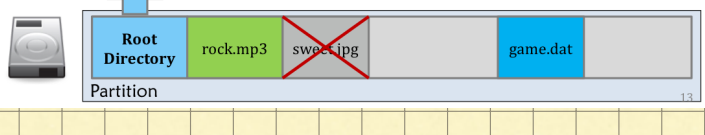
Free space is here

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

Yet, how about file creation?

Filename	Starting Address	Size
rock.mp3	100	1900
~~sweet.jpg~~	~~2001~~	~~1234~~
game.dat	5000	1000

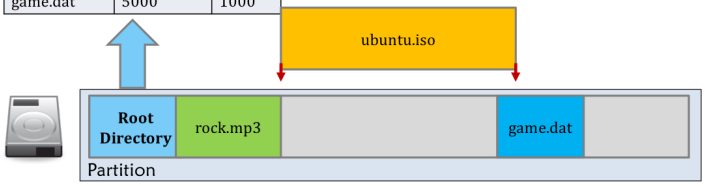
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	5000	1000



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

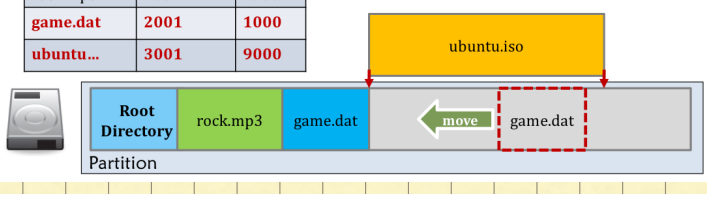
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	5000	1000



创建新文件时会因 fragmentation 不够大.

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

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



文件移位能解决 external fragmentation

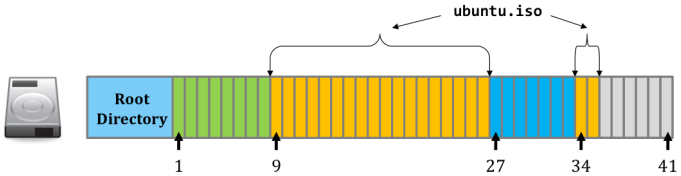
Filename	Starting Address	Size
rock.mp3	100	1900
game.dat	2001	1000
ubuntu...	3001	9000

Growth problem!

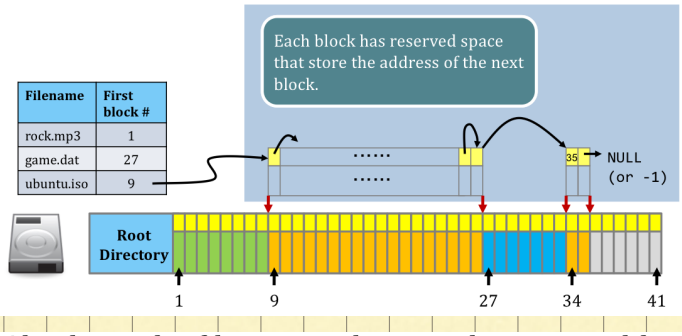
但是增长受限

Linked allocation

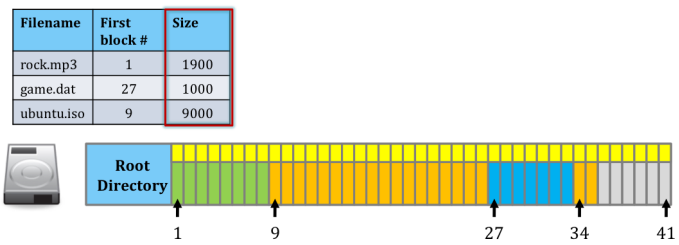
- Let's borrow the idea from the linked list ...
  - Step (1) Chop the storage device into **equal-sized blocks**.
  - Step (2) Fill the empty space in a **block-by-block** manner.



- 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
  - To facilitate "ls –l" that lists the file size of each file
  - (otherwise needs to live counting how many blocks each file has)

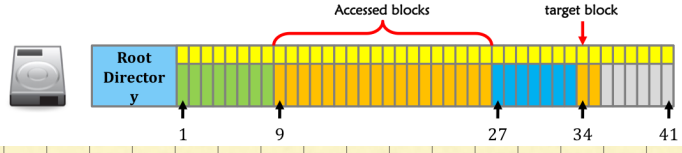


- **Internal Fragmentation**.
  - A file is not always a multiple of the block size.
    - The last block of a file may not be **fully filled**.
    - E.g., a file of size 1 byte still occupies one block.
  - The remaining space will be wasted since no other files can be allowed to fill such space.

文件大小可能不是block大小的整数倍.

- **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**

很难随机访问.



FAT

- Centralize all the block links as File Allocation Table

Task: read "ubuntu.iso" sequentially.

Filename	First block #	Size
rock.mp3	1	1900
game.dat	27	1000
ubuntu.iso	9	9000

Step 1. Read the root directory and retrieve the **first block number**.

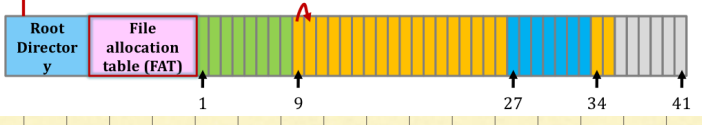
Step 2. Read the FAT to determine the location of next block.

Step 3. After reading the 2nd block, the process continues. Note that the blocks **may not be contiguously allocated**.

Step 4. The process stops until the FAT says the next block # is -1.

File allocation table (FAT)

Block #	1	...	7	8	9	...	26	27	...	32	33	34	35	...	41
Next Block #	2	...	8	-1	10	...	34	28	...	33	-1	35	-1	...	0



FAT

- Start from floppy disk and DOS
- On DOS, a block is called as a '**cluster**'
- E.g., FAT12
  - 12-bit cluster address
  - Can point up to 2^{12} = 4096 blocks

Block #	1	...
Next Block #	2	...

?? bits

	FAT12	FAT16	FAT32
Cluster address length	12 bits	16 bits	28 bits
Number of clusters	2^{12} (4,096)	2^{16} (65,536)	2^{28}

MS reserves 4 bits (but nobody eventually used those)

- Size of a block (cluster):

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

block **size: 32KB** | block **address: 28 bits**

E.g.,

File system size.

$$(32 \times 2^{10}) \times 2^{28} = 2^5 \times 2^{10} \times 2^{28} = 2^{43} \ (8\ TB)$$

*** but MS deliberately set its formatting tool to format it up to 32GB only to lure you to use NTFS**

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.

Reserved sectors

A FAT partition

FAT2是FAT1的备份.

FAT series – directory traversal

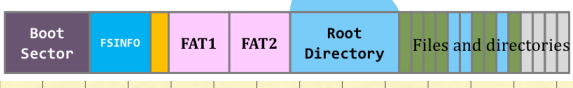
Step (1) Read the directory file of the root directory starting from **Cluster #2**.

"`C:\windows`" starts from Cluster #123.

```
c:\> dir c:\windows
……
06/13/2007  1,033,216    gamedata.dat
08/04/2004     69,120    notepad.exe
……
c:\> _
```

A directory entry

Cluster #2		
Filename	Attributes	Cluster #
.	……	……
..	……	……
……	……	……
windows	……	123



Step (2) Read the directory **file** of the "`C:\windows`" starting from **Cluster #123**.

```
c:\> dir c:\windows
……
06/13/2007  1,033,216    gamedata.dat
08/04/2004     69,120    notepad.exe
……
c:\> _
```

Cluster #123		
Filename	Attributes	Cluster #
.	……	……
..	……	……
……	……	……
notepad.exe	……	456

FAT series – 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

Note. This is the 8+3 naming convention.

8 characters for name +
3 characters for file extension

So, what is the largest size of a FAT32 file?

4G - 1 bytes

Bounded by the file size attribute!

Why "– 1"?

- Imagine 3 bits: 000, 001, …, 110, 111
- Largest number is 111 = 2^3-1
- i.e., we also need to represent "0 bytes"

34

FAT series – 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

Directory file
LFN …
LFN #2
LFN #1
Normal Entry

Each LFN entry represents 13 characters in Unicode, i.e., 2 bytes per character.
Yet, the sequence is upside-down!

A normal directory entry is **still** there.

Normal directory entry vs 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)

This is the sequence number, and they are arranged in descending order.

The terminating directory entry has the sequence number OR-ed with 0x40.

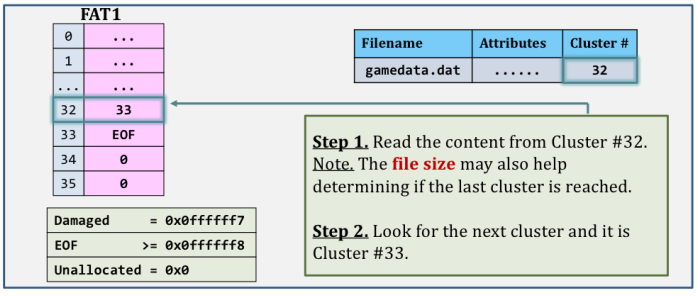
Directory file
LFN #3: "m_cou" "rse.tx" "t"
LFN #2: "erati" "ng_sys" "te"
LFN #1: "I_lov" "e_the_" "op"
Normal Entry

```
LFN #3   43 6d 005f 0063 006f 0075 000f 0040 7200  Cm._.c.o.u...@r.
         7300 6500 2e00 7400 7800 0000 7400 0000  s.e...t.x...t...

LFN #2   02 65 0072 0061 0074 0069 000f 0040 6e00  .e.r.a.t.i...@n.
         6700 5f00 7300 7900 7300 0000 7400 6500  g._.s.y.s...t.e.

LFN #1   01 49 005f 006c 006f 0076 000f 0040 6500  .I._.l.o.v...@e.
         5f00 7400 6800 6500 5f00 0000 6f00 7000  _.t.h.e._...o.p.

Normal   495f 4c4f 5645 7e31 5458 5420 0064 b99e  I_LOVE~1TXT .d..
         773d 773d 0000 b99e 773d 0000 0000 0000  w=w=....w=......
```

- 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**.

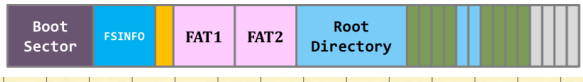
reading a file

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



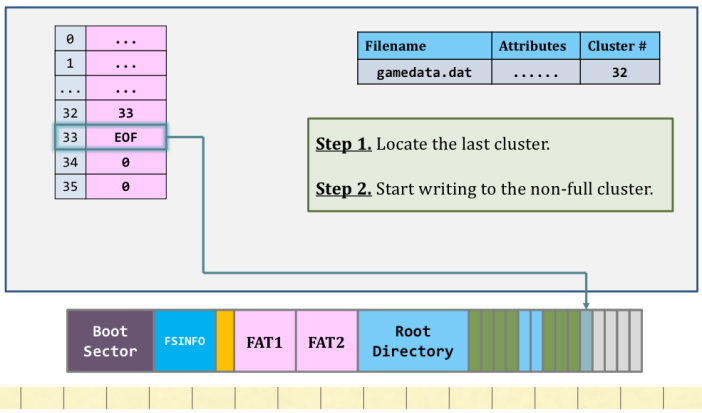
Step 3. Since the FAT has marked "EOF", we have reached the last cluster of that file.

Note. The file size help determining **how many bytes to read** from the last cluster.

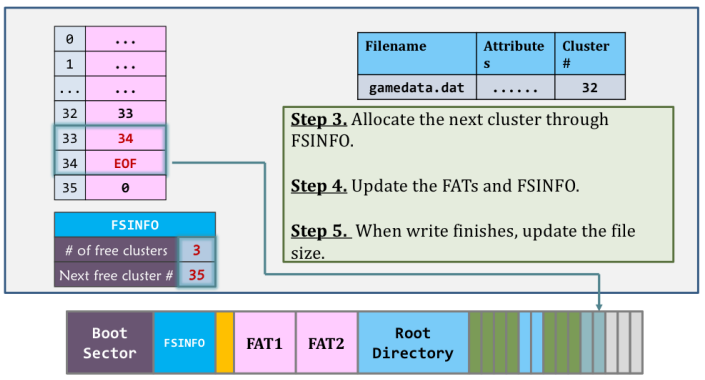


writing a file

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



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

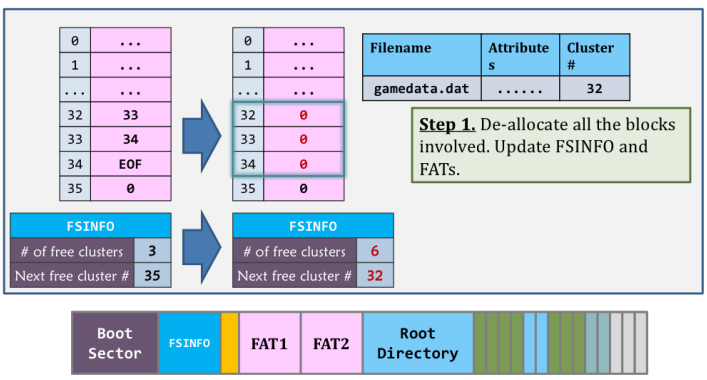


The search for the next free cluster is a circular, next-available search.

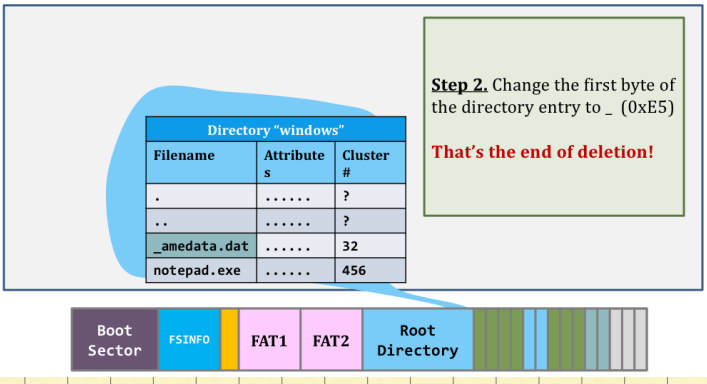
Good for **spatial locality**

delete a file

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



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



Really delete a file

- Can you see that: **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.
- Any way(s) to delete a file **securely**?

被删除的文件实际上并未从文件系统中移除.

How to recover a deleted file ?

- If you really care about the deleted file, then…
  - **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?

FAT series – conclusion

- Space efficient:
  - 4 bytes overhead (FAT entry) per data cluster.
- Delete:
  - Lazy delete efficient
  - Insecure
    - designed for single-user 20+ years ago
- Deployment: (FAT32 and FAT12)
  - It is everywhere: CF cards, SD cards, USB drives
- Search:
  - Block addresses of a file may scatter discontinuously
  - To locate the 888-th block of a file?
    - Start from the first FAT entry and follow 888 pointers
- The most commonly used filesystem in the world

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

Summary

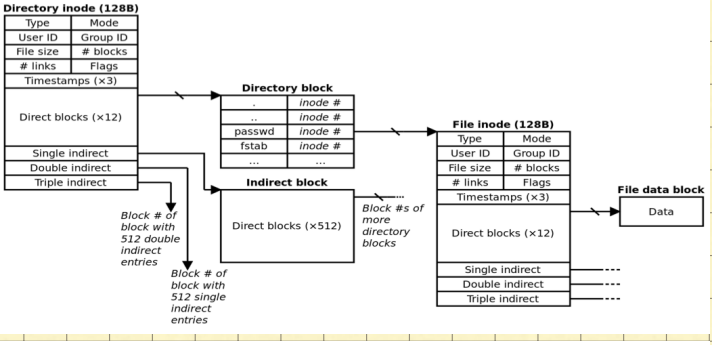
- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- File Allocation Table (FAT) Scheme
  - Linked-list approach
  - Very widely used: Cameras, USB drives, SD cards
  - Simple to implement, but poor performance and no security

Unix File System

- Original iNode format appeared in BSD 4.1
  - Berkeley 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 the file
  - Rather than in the directory that points to it
- Scalable directory structure

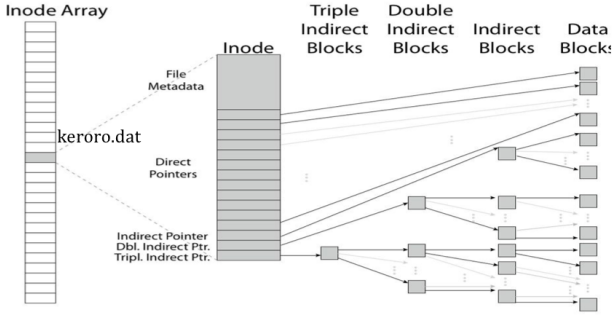
iNode

- All pointers of a file are located together
  - **VS. FAT: pointers of a file are scattered**
- One directory/file has one iNode



每个directory/file都有一个iNode，相当于File Header

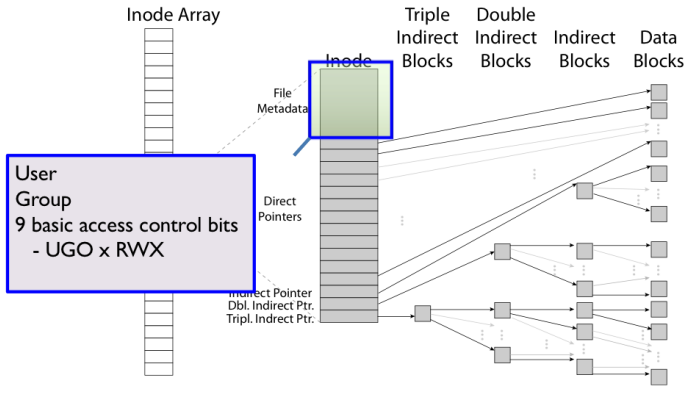
- iNode Table is an array of iNodes
- Pointers are unbalanced tree-based data structures



只有叶子节点存data，其它节点都存指向下一层的pointer.

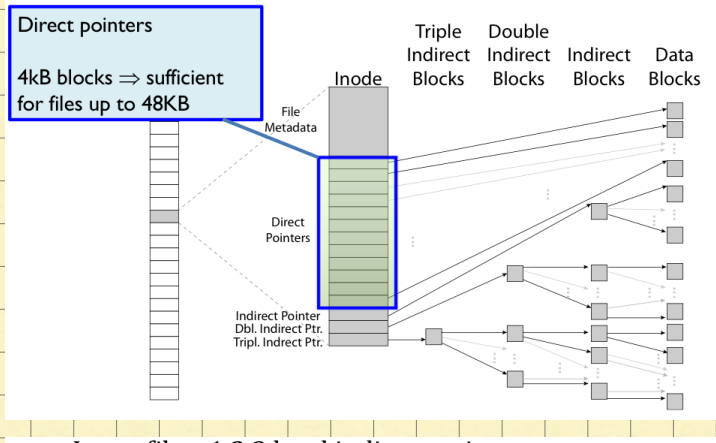
File Attributes

- iNode metadata

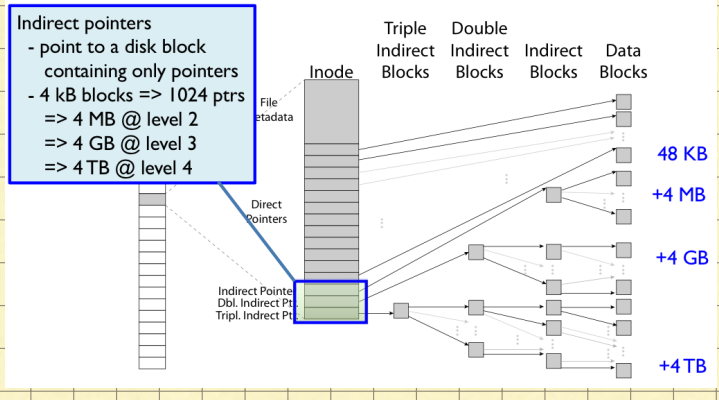


Data Storage

- Small files: 12 pointers direct to data blocks



- Large files: 1,2,3 level indirect pointers



Index node – file size

Reminder: Max file size != FS size

Number of direct blocks	12
Number of indirect blocks	1
Number of double indirect blocks	1
Number of triple indirect blocks	1
Block size	2^x bytes
Address length	4 bytes

contains "2^x / 4" addresses

$$12 \times 2^x + 1 \times 2^x/4 \times 2^x + 1 \times (2^x/4)^2 \times 2^x + 1 \times (2^x/4)^3 \times 2^x$$

File size = number of data blocks * Block size

Block size 2^x	Max size
1024 bytes = 2^{10}	approx. 16 GB
4096 bytes = 2^{12}	approx. 4 TB

Ext

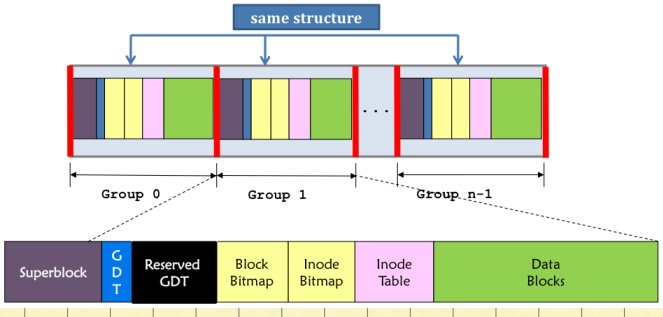
- The latest default FS for Linux distribution is the **Fourth Extended File System**, **Ext4** for short.

- For Ext2 & Ext3:
  - Block size: 1,024, 2,048, or 4,096 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

Ext 2/3 – Block groups

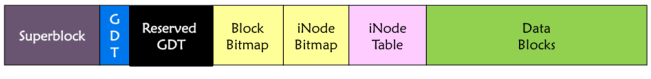
- The file system is divided into **block groups** and every block group has the **same structure**



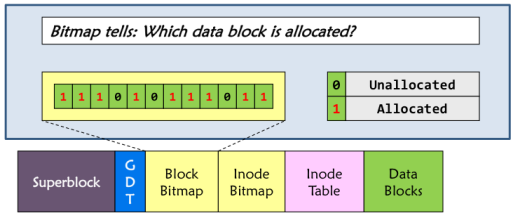
Ext 2/3 – FS layout

- Layout of one block group is as follows:

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.



Ext 2/3 – Block Bitmap & iNode Bitmap

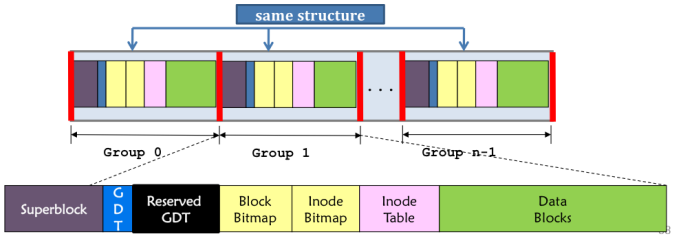


iNode Bitmap
- A bit string that represents if an iNode (index-node) is allocated or not

➔ implies that the **number of files in the file system is fixed**!

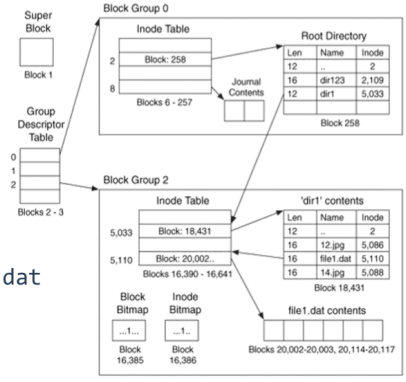
Ext 2/3 – Block groups

- Why having groups?
- For **(1) performance** and **(2) reliability**
  - (1) Performance: spatial locality.
    - Group iNodes and data blocks of related files together
  - (2) Reliability: superblock and GDT are **replicated** in each block group (yes, very reliable!)



Linux Example : Ext2/3 Disk Layout

- Disk divided into block groups
  - Each group has two block-sized bitmaps (free blocks/inodes)
  - Block sizes settable at format time: 1K, 2K, 4K, 8K...
  - Provides locality



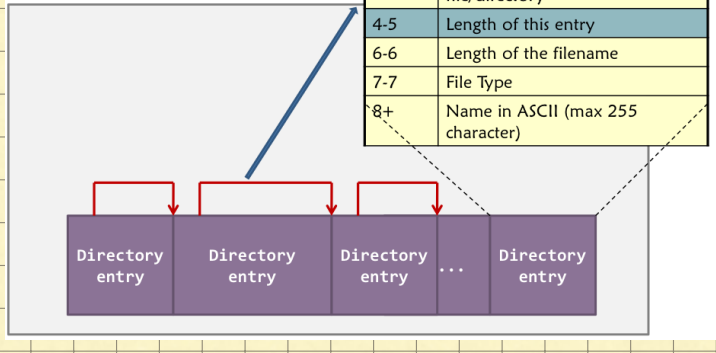
- Example: create a `file1.dat` under `/dir1/` in Ext3

Ext 2/3 – 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

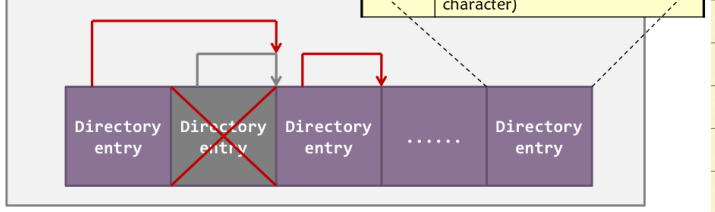
The locations of the data blocks are stored in the inode.

Directory entry 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 is just an update of the "entry length" of the previous entry.

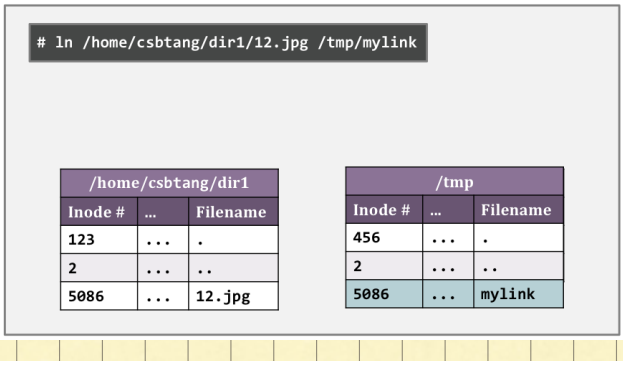
Directory entry structure	
Bytes	Value
0-3	INode number of that file/directory
4-5	Length of this entry
6-6	Length of the filename
7-7	File Type
8+	Name in ASCII (max 255 character)



Ext2/3 – link file: What is a hard link

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

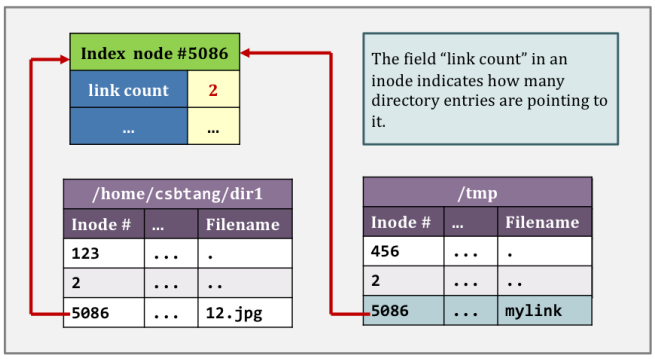
硬连接是 directory entry



- That **file can be accessed through two different pathnames**.

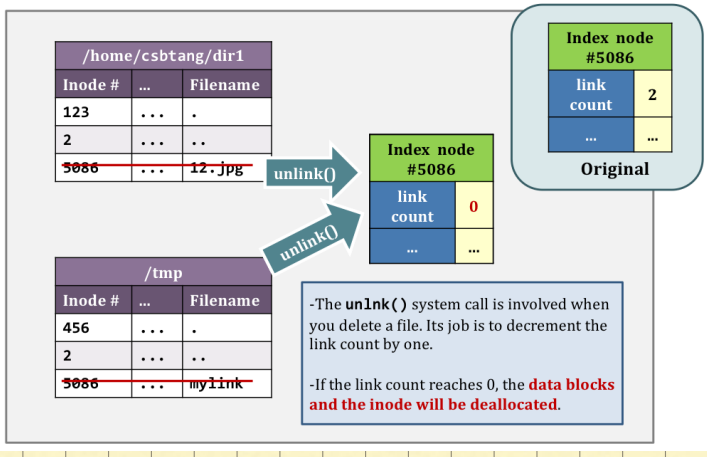
硬连接能直接打开对应 link 的文件的 iNode

有 link count 记录有多少硬连接指向该 iNode



removing file and link count

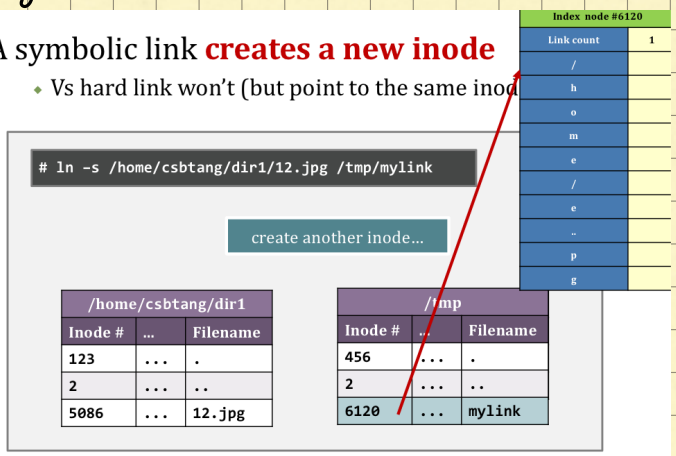
link count 为 0 时该 iNode 会被复用.



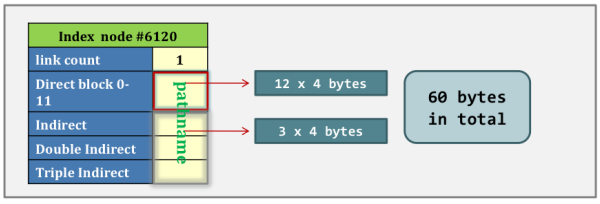
Symbolic link

- A symbolic link **creates a new inode**
  - Vs hard link won't (but point to the same inod

软连接会新建一个 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 long pathname otherwise



软连接的iNode中原本的direct/indirect block pointer用于存放路径名.

若路径名过长,则用一个普通iNode和一个用于存放路径名的direct data block.

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

Property/Action		Symbolic link	Hard link
When the link is deleted		Target remains unchanged	Reference counter is decremented; when it reaches 0, the target is deleted
When target is moved		Symbolic link becomes invalid	Hard link remains valid
Relative path		Allowed	N/A
Crossing filesystem boundaries		Supported	Not supported (target must be on same filesystem)
Windows	For files	Windows Vista and later[20] (administrator rights required)	Yes
	For folders		No
Unix	For files	Yes	Yes
	For directories	Yes	Partial[21]

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!!

File System Summary

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
- 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

- 4.2 BSD Multilevel index files
  - 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
- File layout driven by freespace management
  - Integrate freespace, iNode table, file blocks and dirs into block group
- Deep interactions between memory management, file system, sharing
  - `mmap()`: map file or anonymous segment to memory