Operating Systems

Associate Prof. Yongkun Li
中科大-计算机学院 副教授
http://staff.ustc.edu.cn/~ykli

Ch10, part2

Details of Ext2/3 File System

Trivia

- Extended File System (Ext2/3/4)
  - Follow index-node allocation
  - Primary FS for Linux distribution
    - Ext4 was merged in the Linux 2.6.28 and released in 2008
  - Backward-compatible
  - For simplicity, we focus on Ext2/3
    - Features of Ext2/3/4
    - https://ext4.wiki.kernel.org/index.php/Main_Page
    - http://e2fsprogs.sourceforge.net/ext2.html

Details of Ext2/3

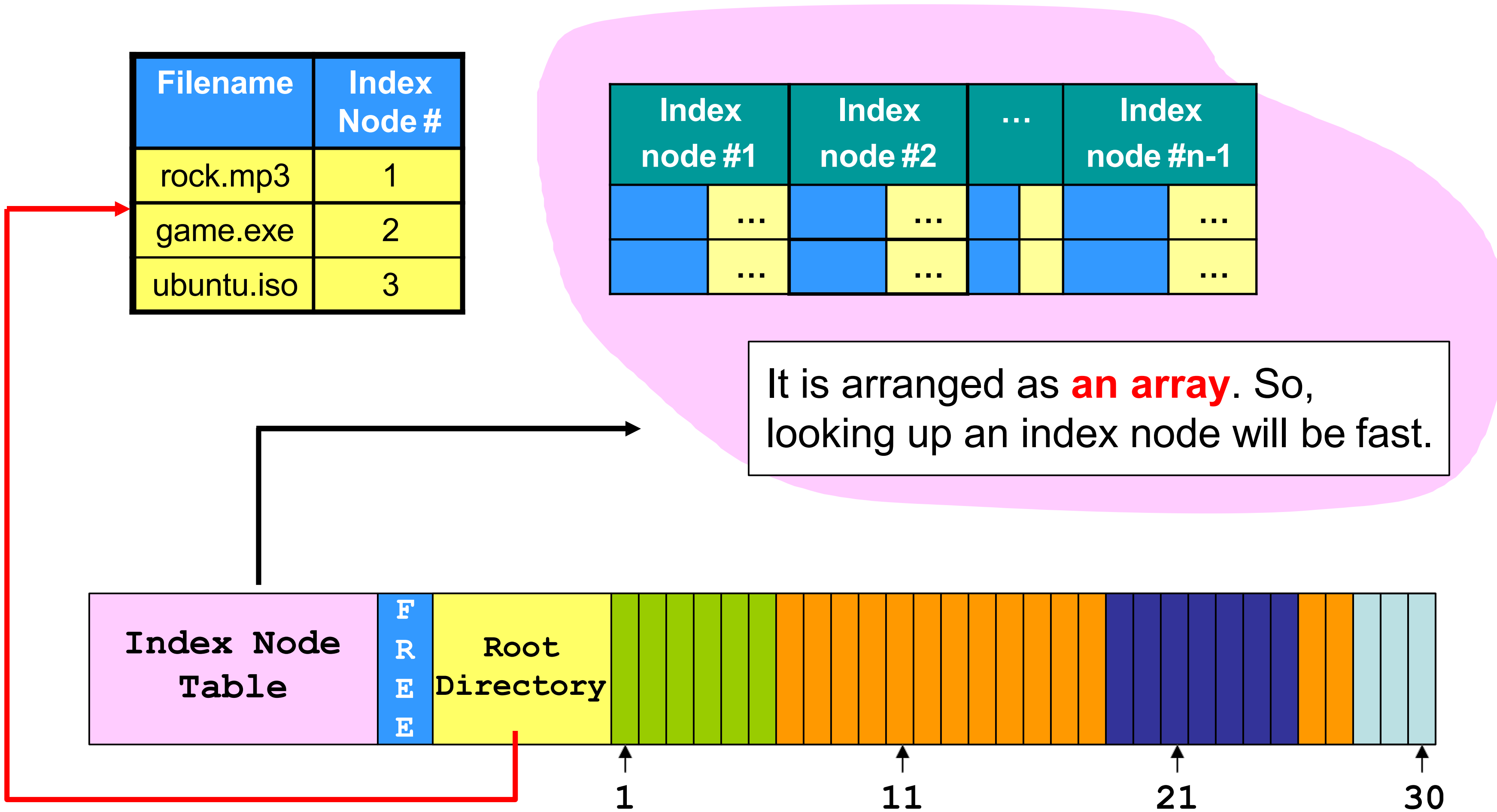
- Layout
- Inode and directory structure
- Link file
- Buffer cache
- Journaling
- VFS

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- Buffer cache
- Journaling
- VFS

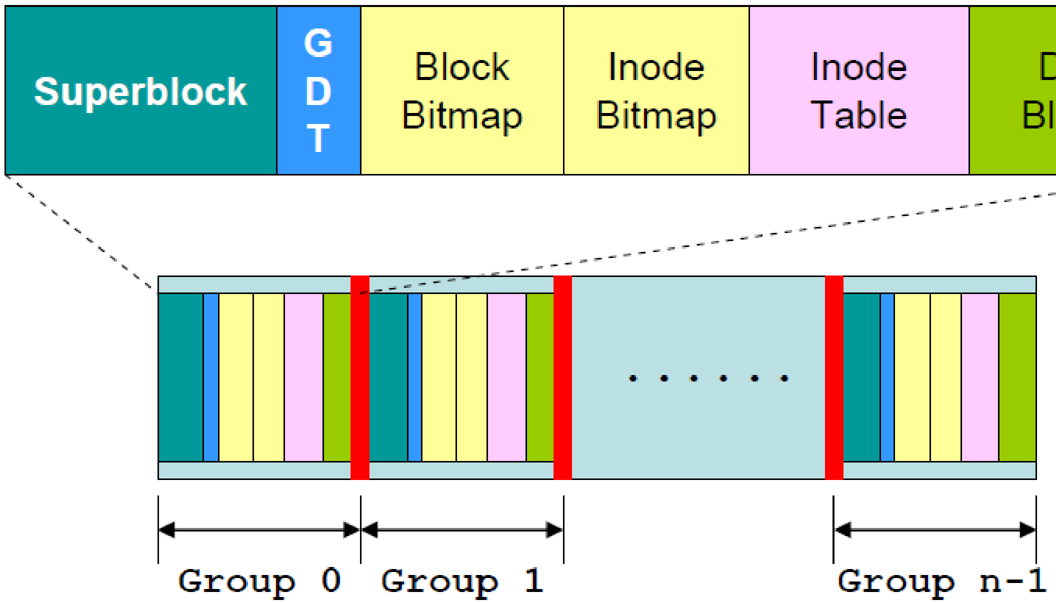
Index-node allocation

- Ext2/3 file systems follow the index-node allocation



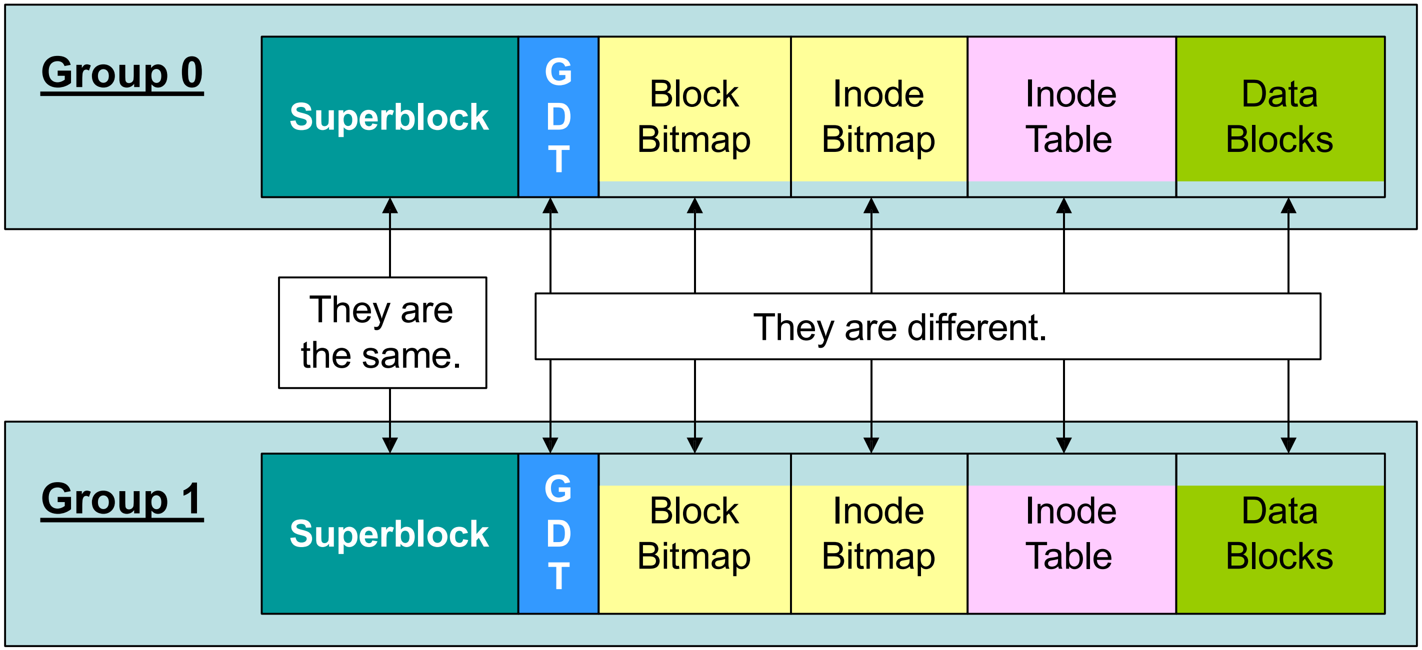
Specific Layout

- The file system is not that simple…
  - it is divided into groups, and …
  - every group has the same structure.



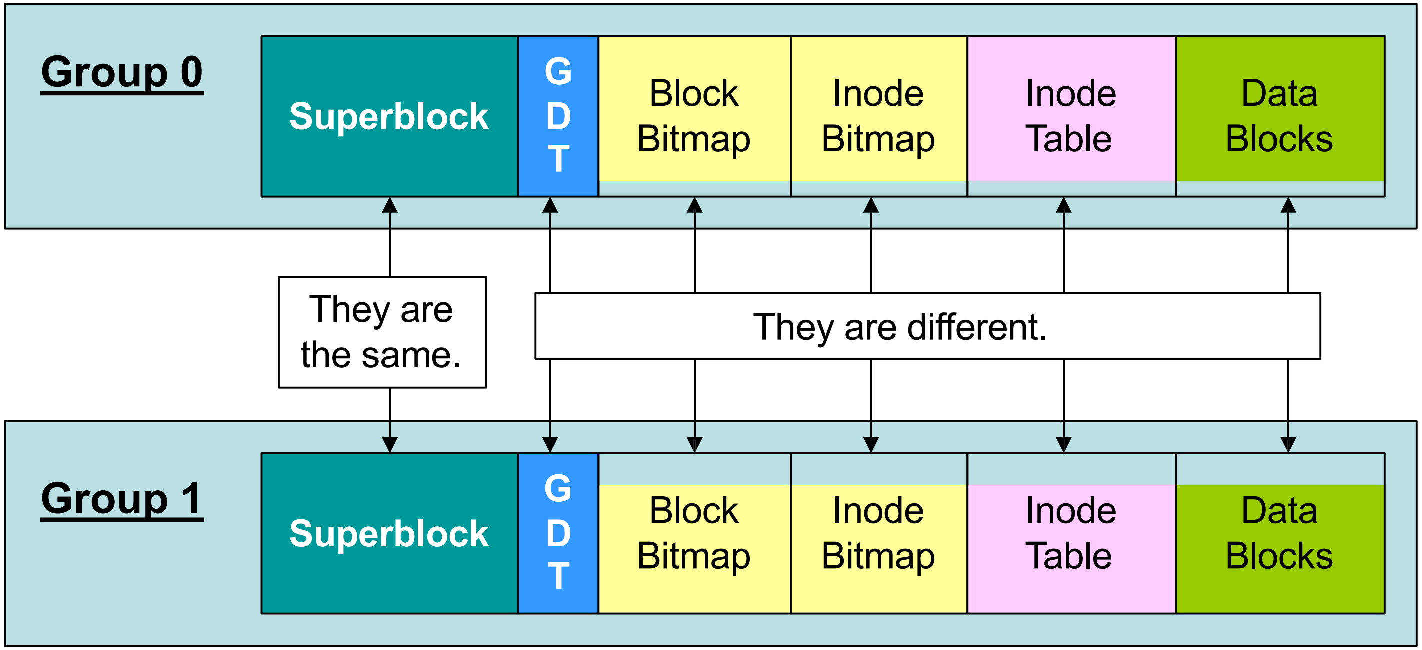
Specific Layout

- The file system is not that simple…
  - it is divided into groups, and …
  - every group has the same structure.



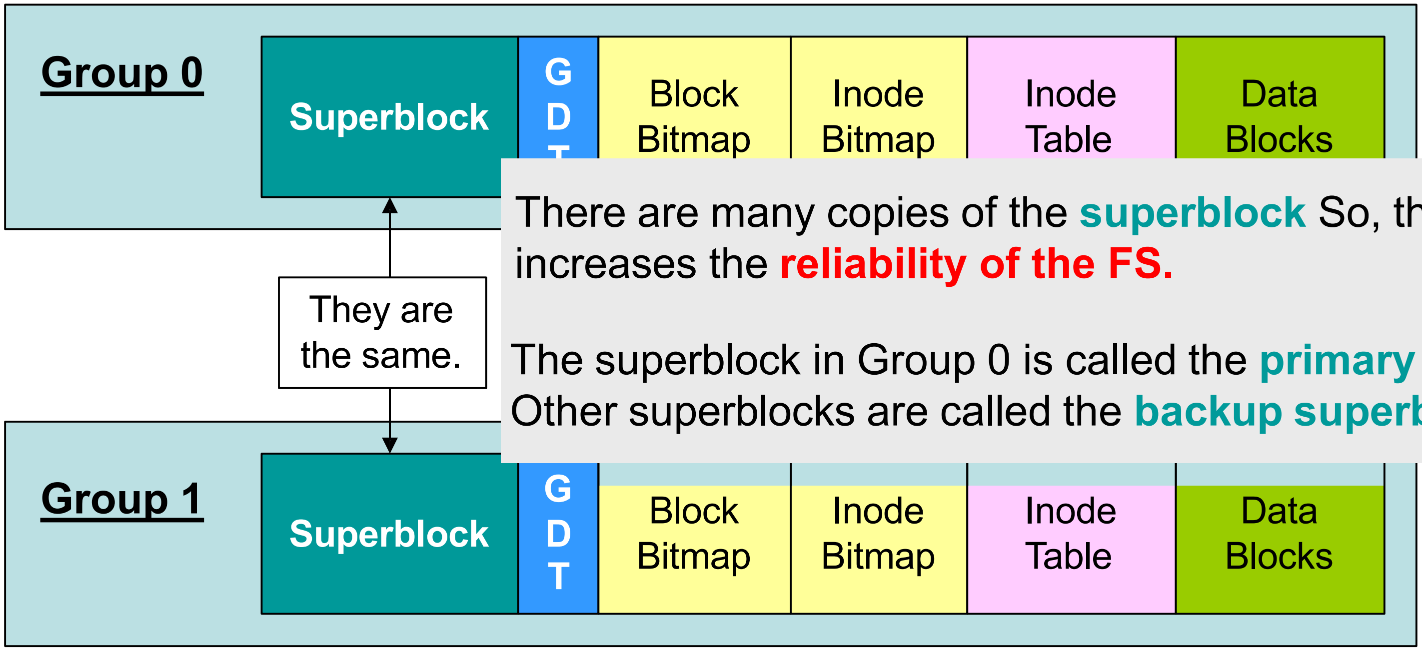
Specific Layout

- Why doing so?
  - This is for **reliability and performance**.



Specific Layout

- Why doing so?
  - For **reliability…**

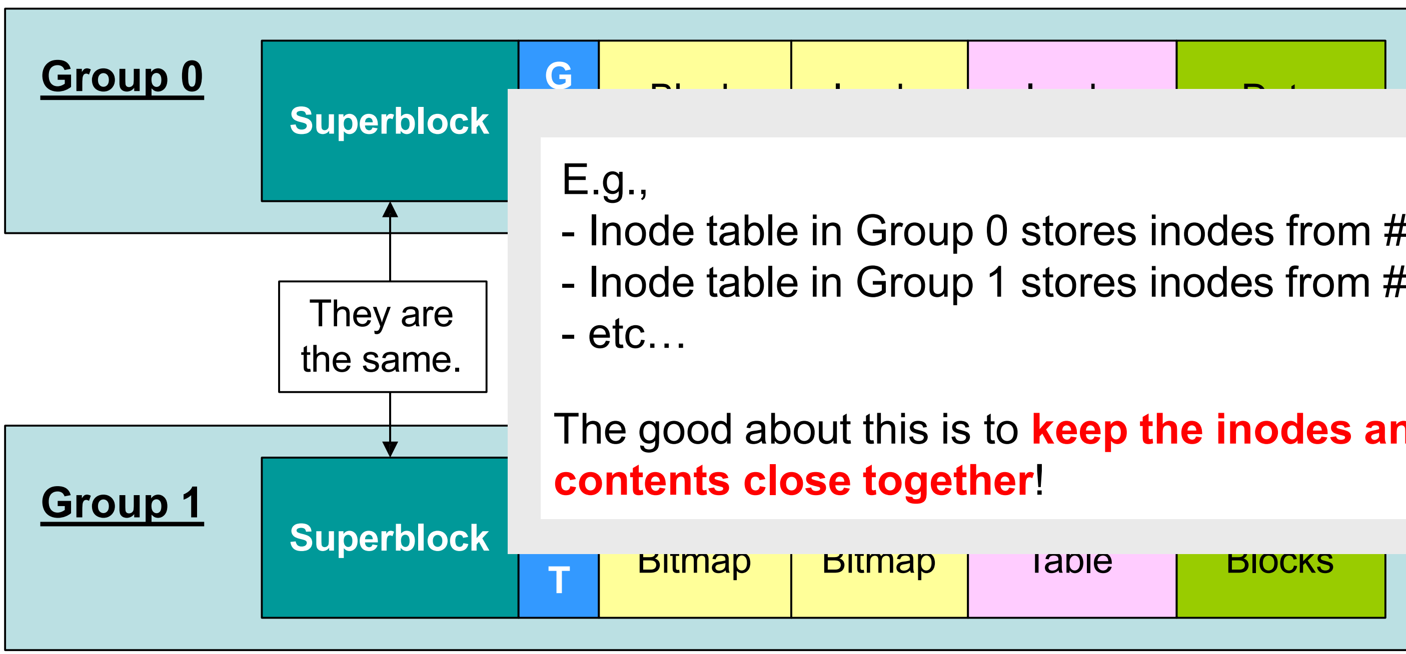


There are many copies of the **superblock** So, this increases the **reliability of the FS.**

The superblock in Group 0 is called the **primary superblock**. Other superblocks are called the **backup superblock**.

Specific Layout

- Why doing so?
  - –For **performance…**



E.g.,
- Inode table in Group 0 stores inodes from #1 to #100;
- Inode table in Group 1 stores inodes from #101 to #200;
- etc…

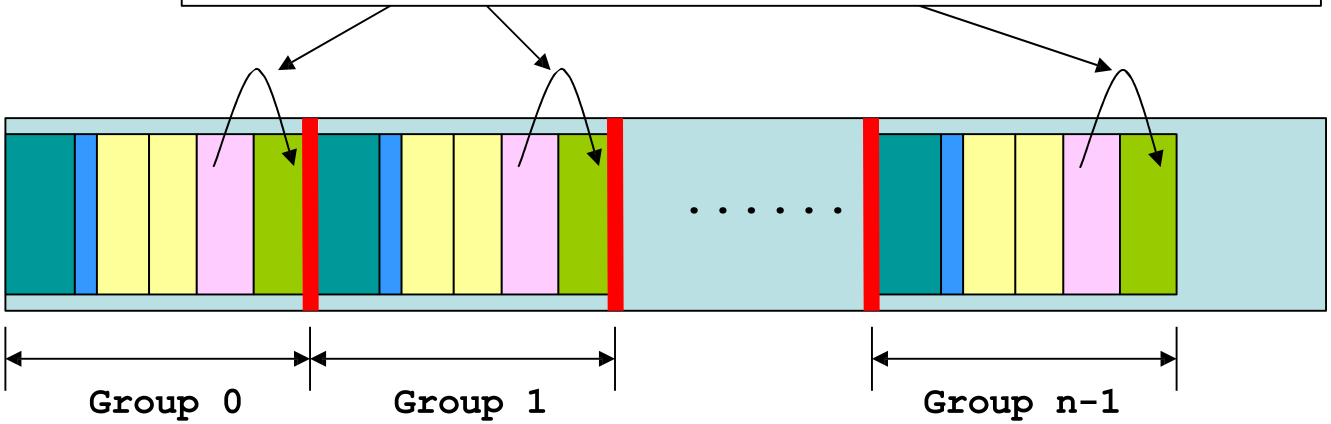
The good about this is to **keep the inodes and the file contents close together**!

Specific Layout

- Why doing so?
  - For **performance…**

The inodes in a particular group will *usually* refer to the data blocks in the same group.

So, this keeps them close together in a physical sense. The storage device may be able to locate the data in a faster manner. (*Remember the* ***principle of locality****?*)



Layout in Each Group

Superblock	GDT	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

Superblock	Stores FS specific data.

Total number of inodes in the system.
Total number of blocks in the system.
Number of reserved blocks
Total number of free blocks.
Total number of free inodes.
Location of the first block.
The size of a block.

Layout in Each Group

Superblock	**G D T**	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

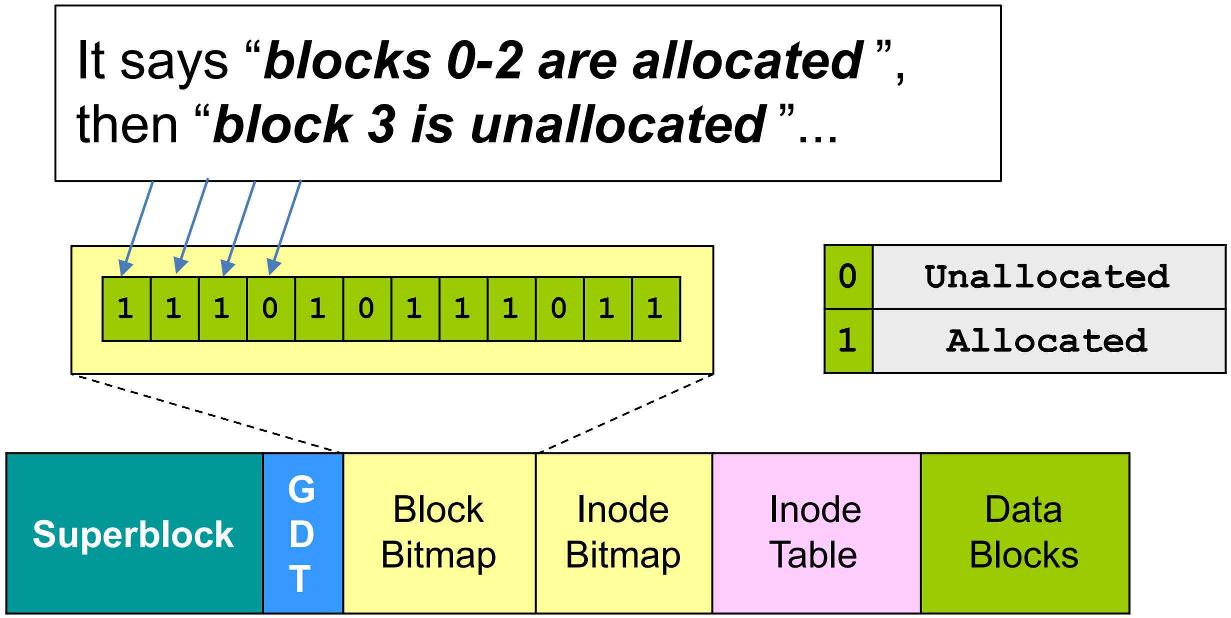
Superblock	Stores FS specific data. E.g., the total number of blocks, etc.
GDT – Group Descriptor Table	It stores: -The **starting block numbers** of the block bitmap, the inode bitmap, and the inode table. - Free block count, free inode count, etc…

Inode Table	An array of inodes ordered by the inode #.
Data Blocks	An array of blocks that stored files.

Block Bitmap	A **bit string** that represents if **a block** is allocated or not.
Inode Bitmap	A **bit string** that represents if **an inode** is allocated or not.

Layout in Each Group

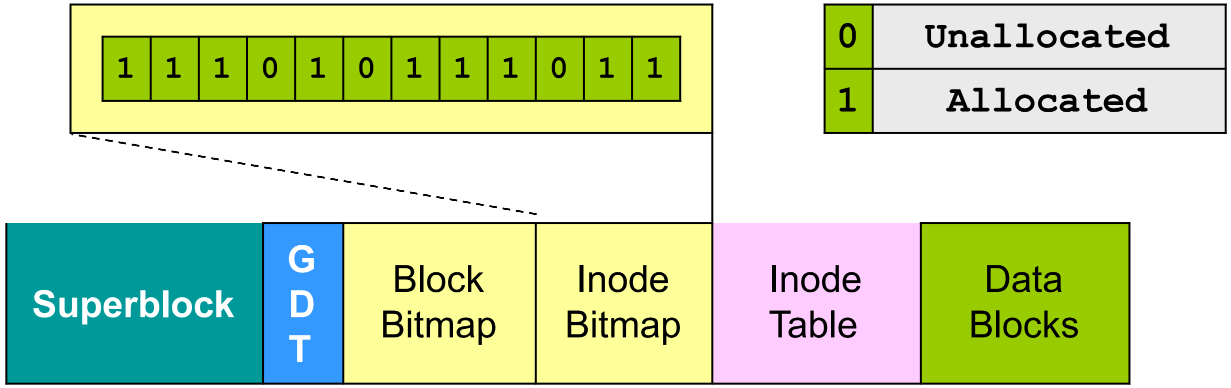
- What is a **block bitmap**?
  - A sequence of bits indicates **the allocation of the blocks**.



Layout in Each Group

- Then, what is an **inode bitmap**?
  - A sequence of bits indicates **the allocation of the inodes**.
  - This implies that…

The **number of files** in the file system is fixed!

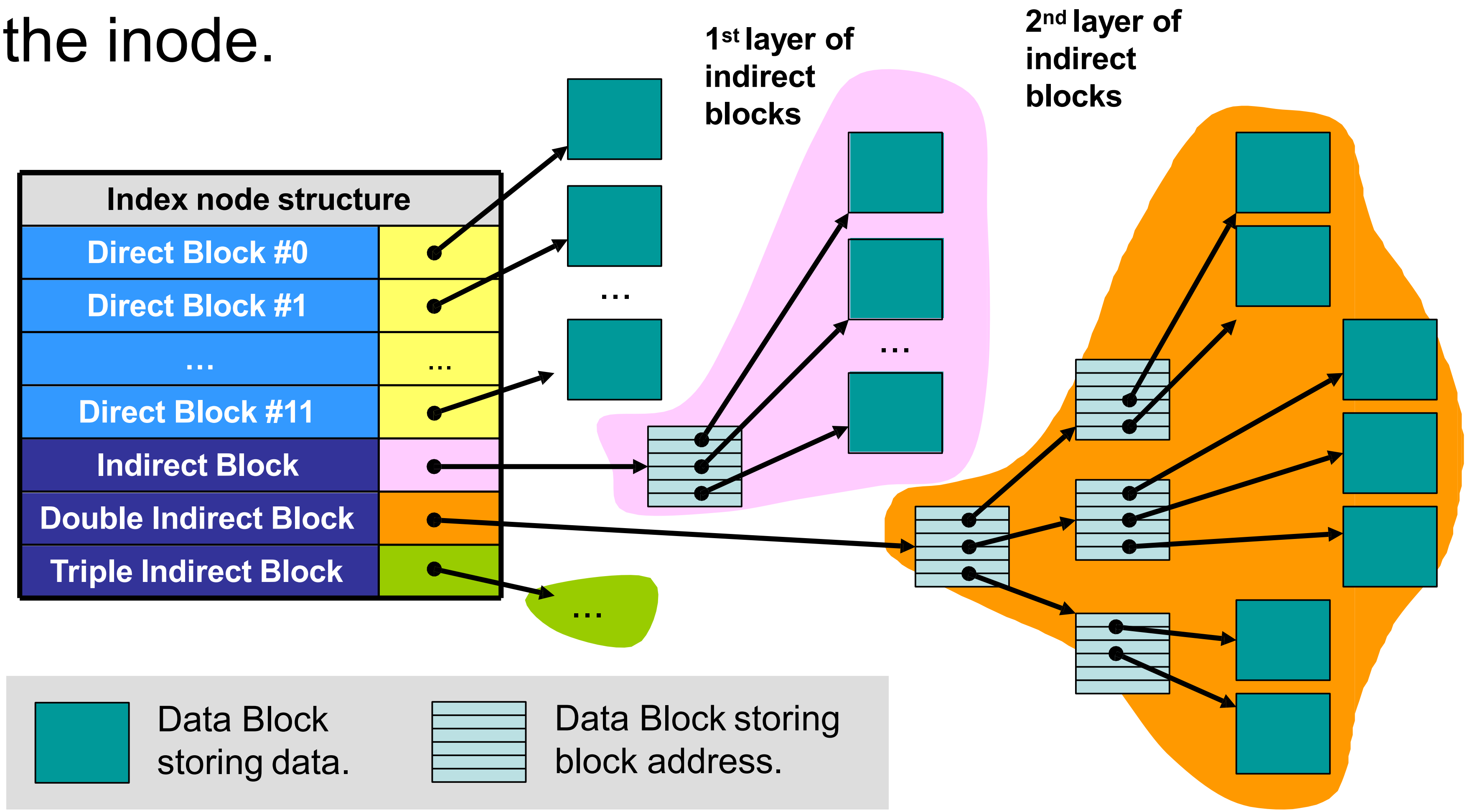


Details of Ext2/3

- Layout
- **Inode and directory structure**
- Link file
- Buffer cache
- Journaling
- VFS

Inode Structure

- We know that…
  - The locations of the data blocks of a file are stored in the inode.



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
…	…
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

What are stored in inode besides block addresses?

An inode is the structure that stores every information about a file.

The locations of the data blocks

More details: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Inode_Table

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
…	…
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

What is the maximum file size supported?

$2^{64} - 1$

$= 16 \times 2^{30}$ Gbytes – 1 byte

Is this really the case?

Remember the dominating factor: 2^{4x-6}

Block size	File size
1024B = 2^{10}	~16 Gbytes
4096B = 2^{12}	~4 Tbytes

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
…	…
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

What is link count?

We will talk about it later

Where is the file name?

Let us take a look at the directory structure

Directory Structure

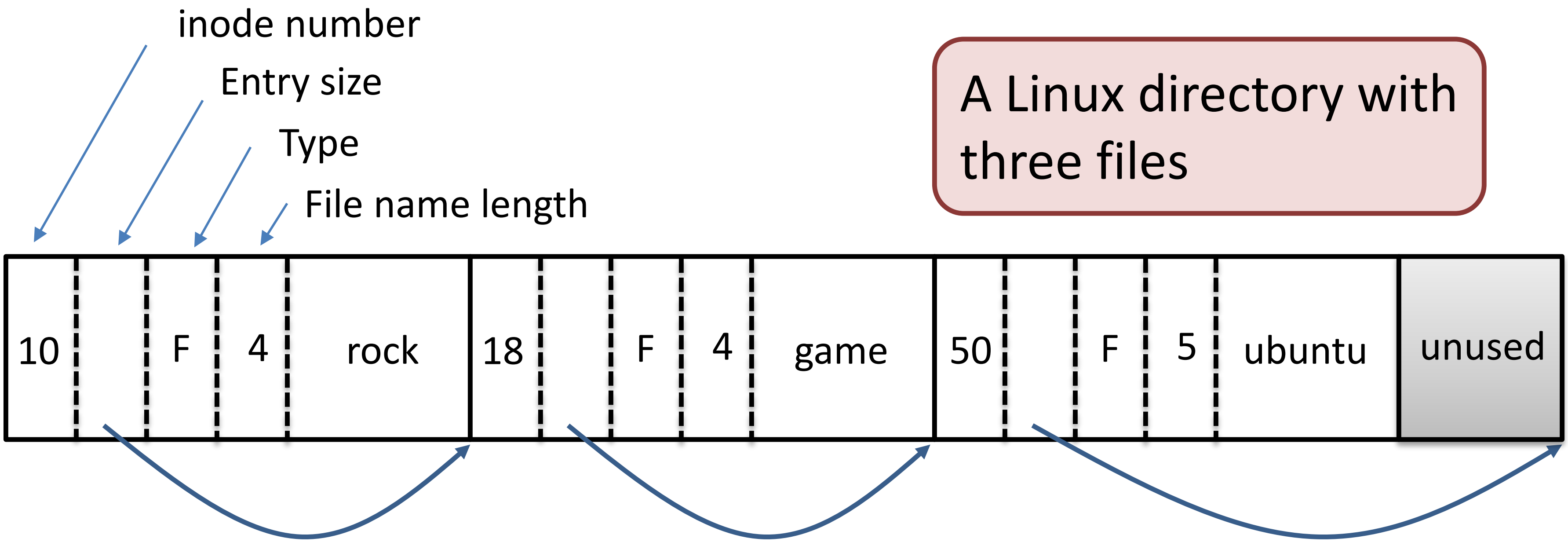
Filename	Index Node #
rock.mp3	1
game.exe	2
ubuntu.iso	3

The directory entry stores the **file name** and **the inode #**.

```
int main(void) {
    DIR * dir;
    struct dirent *entry;

    di

    wh



    }

    cl
    re
}
```

```
struct dirent {
    ino_t              d_ino;       // inode number
    off_t              d_off;       // offset to the next dirent
    unsigned short     d_reclen;    // record length
    unsigned char      d_type;      // file type
    char *             d_name;      // file name
}
```

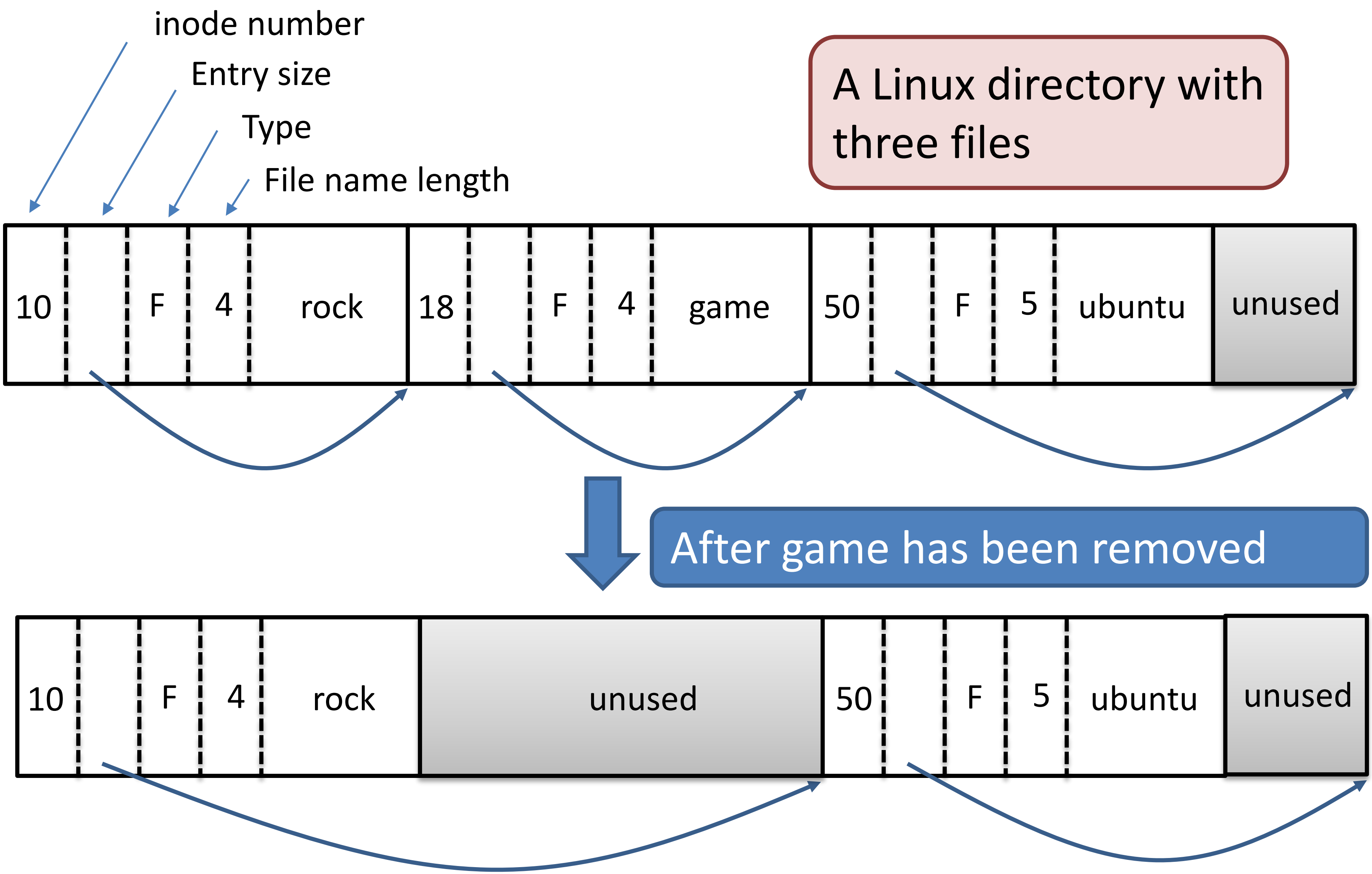
Directory Structure



A Linux directory with three files

```
struct dirent {
    ino_t               d_ino;        // inode number
    off_t               d_off;        // offset to the next dirent
    unsigned short      d_reclen;     // record length
    unsigned char       d_type;       // file type
    char *              d_name;       // file name
}
```

Directory Structure



Accessing Directory File

- How to access directory file?

Note: **`opendir()`**, **`readdir()`**, and **`closedir()`** are library function calls.

```
int main(void) {
     DIR * dir;
     struct dirent *entry;

     dir = opendir(“/”);

     while ( (entry = readdir(dir))  != NULL)  {
       // print the directory name
         printf(“%s\n”, entry->d_name);
     }

     closedir(dir);
     return 0;
}
```

Open the directory file.

Read the directory entries one by one until there is not further entries.

Close the directory file.

Details of Ext2/3

- Layout
- Inode and directory structure
- **Link file**
- Buffer cache
- Journaling
- VFS

Link File

- Can we allow a file to have multiple names and be accessed by several paths?
- How to create shortcuts?

Example use in Linux

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
# _
```

```
# ls /dir1/12.jpg
12.jpg
# ln -s /dir1/12.jpg /my_link
# _
```

These are called **hard link** and **symbolic link**

Link File – what is a hard link?

- A **hard link is a directory entry** pointing to an existing file.
  - **No new file content is created!**

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
# _
```

A new directory entry is created.

Directory: /dir1

Inode #	…	Filename
123	…	.
2	…	..
5,086	…	**12.jpg**

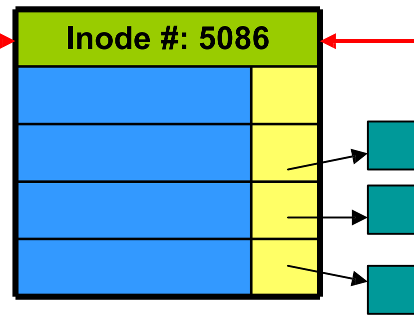
Directory: /

Inode #	…	Filename
2	…	.
2	…	..
5,086	…	**my_link**

Link File – what is a hard link?

- Conceptually speaking, this **creates a file with two pathnames**.

How to maintain this info.



Directory: /dir1

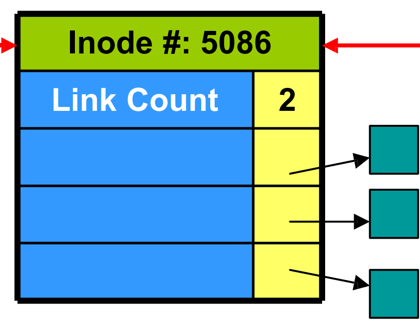
Inode #	…	Filename
123	…	.
2	…	..
5,086	…	**12.jpg**

Directory: /

Inode #	…	Filename
2	…	.
2	…	..
5,086	…	**my_link**

Link File – what is a link count?

- There is a field called **link count** in an inode.
  - It stores the number of directory entries pointing to the inode.



Directory: /dir1

Inode #	…	Filename
123	…	.
2	…	..
5,086	…	**12.jpg**

Directory: /

Inode #	…	Filename
2	…	.
2	…	..
5,086	…	**my_link**

Link File – showing the link counts

- Special hard links
  - The directory "." is a hard link to itself.
  - The directory ".." is a hard link to the parent directory.

What does this large number imply?

```
# ls -l /
total 124
drwxr-xr-x   2 root root  4096 2015-11-15 08:07 bin
drwxr-xr-x   4 root root  4096 2015-11-11 09:25 boot
drwxr-xr-x  17 root root 14520 2015-11-23 17:58 dev
drwxr-xr-x 165 root root 12288 2015-11-23 17:58 etc
drwxr-xr-x   6 root root  4096 2015-06-21 14:23 home
......
```

This implies "/etc" has a lot of sub-directories.

Link File – showing the link counts

- Special hard links
  - The directory "." is a hard link to itself.
  - The directory ".." is a hard link to the parent directory.

- What is the value of the link count, if
  - A **file** is created
  - A **directory** is created

Link File – showing the link counts

- When a regular file is created, the link count is **always 1**

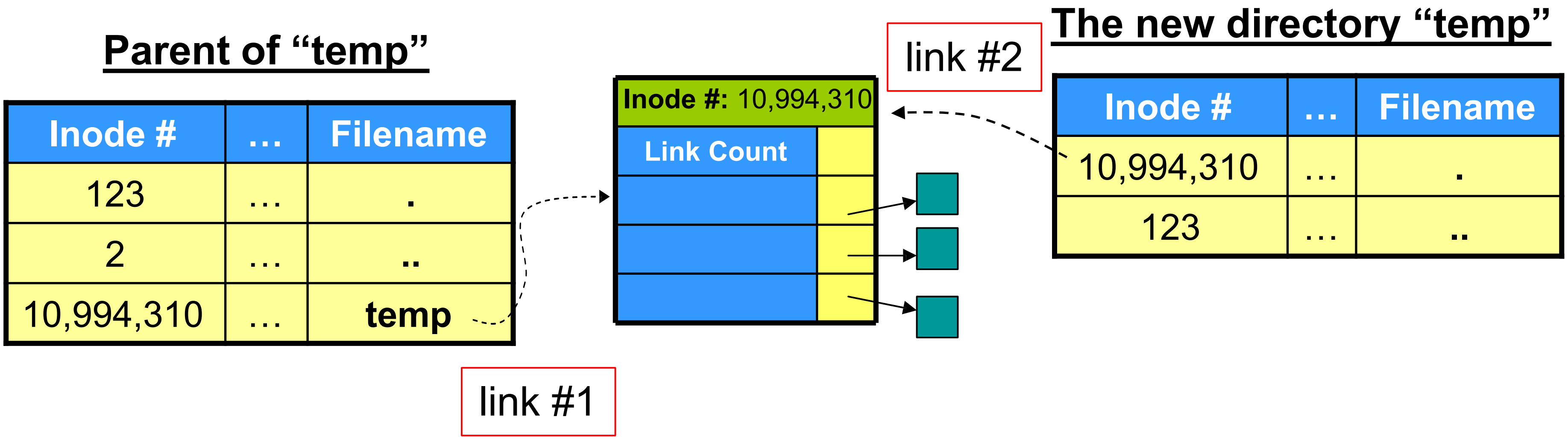
```
# stat Makefile
 File: `Makefile'
  Size: 4552            Blocks: 16         IO Block: 4096   regular file
Device: 801h/2049d      Inode: 30669       Links: 1
......
```

- When a directory is created, **the initial link count is always 2**

```
# mkdir temp
# stat temp
  File: `temp'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 804h/2052d      Inode: 10994310    Links: 2
......
```

Why it is 2

Link File – showing the link counts



- When a directory is created, **the initial link count is always 2**. Why?

```
# mkdir temp
# stat temp
  File: `temp'
  Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 804h/2052d    Inode: 10994310    Links: 2
......
```

Link File – showing the link counts

Parent of “temp”

Inode #	...	Filename
123	...	.
2	...	..
10,994,310	...	temp

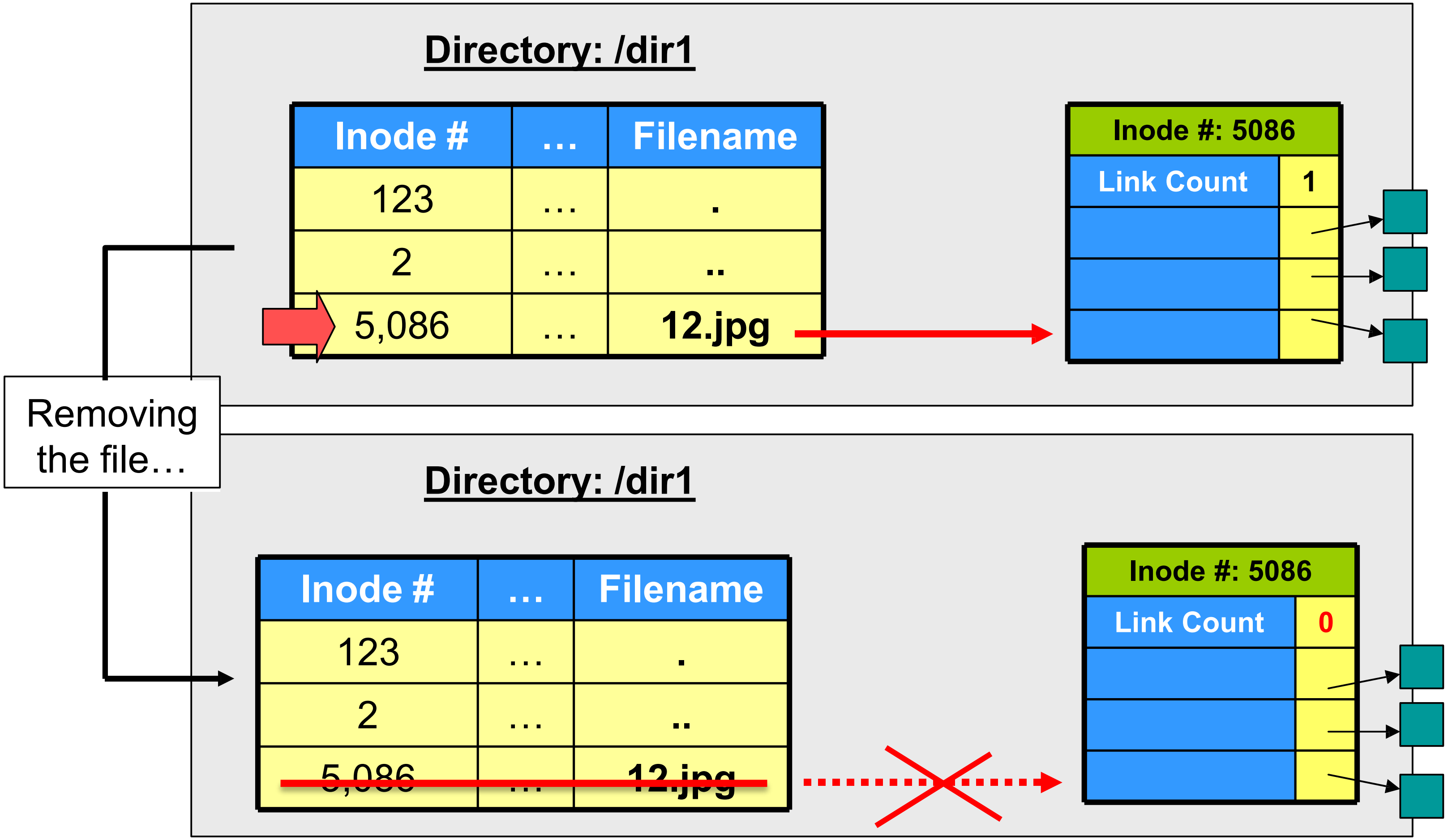
The new directory “temp”

Inode #	...	Filename
10,994,310	...	.
123	...	..

- The hosting directory of the newly creating directory will have its **link count increased by 1**.

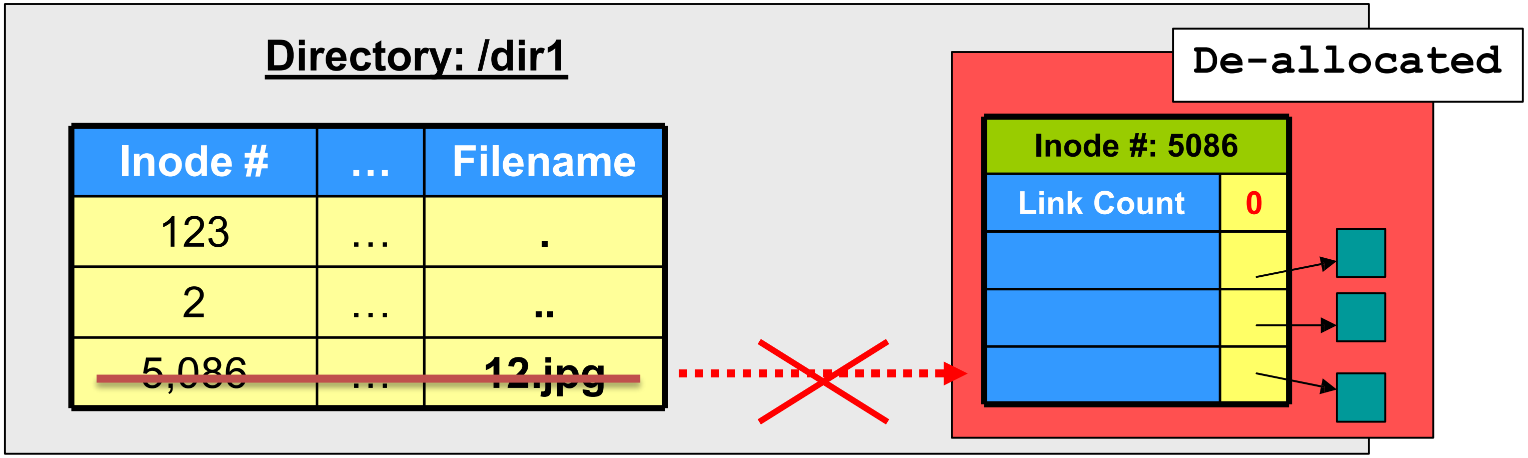
Link File – decrementing the link count?

- How about removing a file?



Link File – decrementing the link count?

- How about removing a file?
  - The system call that removing a file is, therefore, called **`unlink()`**.
    - The **`unlink()`** system call is to decrement the link count by **exactly one**.
    - When the **link count == 0**, the **data blocks** and the **inode** will all be de-allocated by the kernel.



Link File – decrementing the link count?

- Back to the previous hard link example…

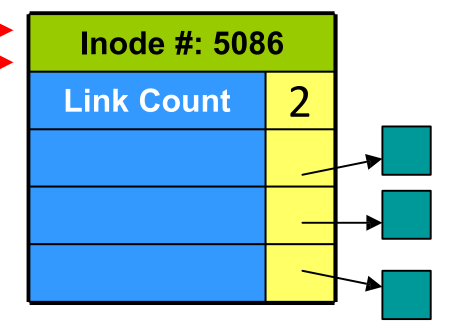
Directory: /dir1

Inode #	…	Filename
123	…	.
2	…	..
5,086	…	**12.jpg**

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
```

Directory: /

Inode #	…	Filename
2	…	.
2	…	..
5,086	…	**my_link**



Link File – decrementing the link count?

- Back to the previous hard link example…

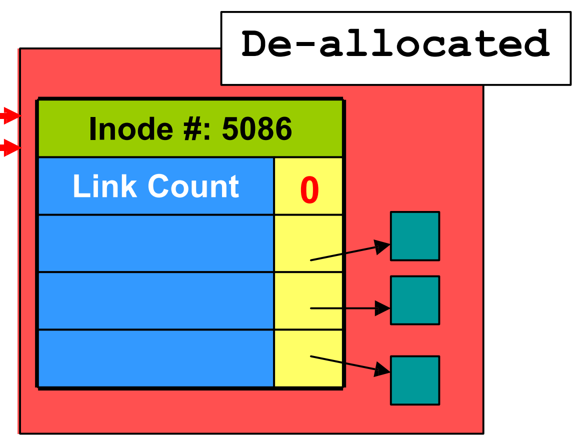
Directory: /dir1

Inode #	…	Filename
123	…	.
2	…	..
~~5,086~~	~~…~~	~~12.jpg~~

Directory: /

Inode #	…	Filename
2	…	.
2	…	..
~~5,086~~	~~…~~	~~my_link~~

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
# rm /dir/12.jpg
# rm /my_link
```



Link File – what is a symbolic link?

- A symbolic link **is a file**.
  - Unlike the hard link, **a new inode is created** for each symbolic link.
  - It stores the pathname (shortcut)

```
# ls /dir1/12.jpg
12.jpg
# ln -s /dir1/12.jpg /my_link
# ls -l /mylink
/mylink -> /dir1/12.jpg
#
```

A new directory entry is created.

Directory: /dir1

Inode #	…	Filename
123	…	.
2	…	..
5,086	…	**12.jpg**

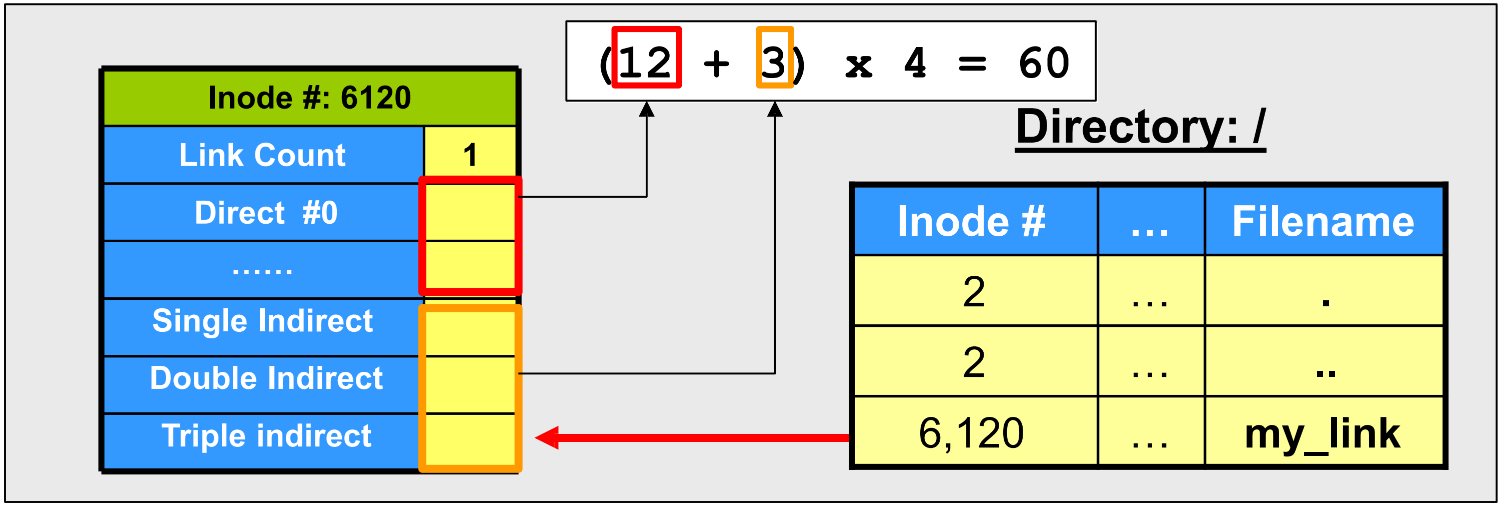
Another inode

Directory: /

Inode #	…	Filename
2	…	.
2	…	..
6,120	…	**my_link**

Link File – what is a symbolic link?

- How to store the target path?
  - If the pathname is less than 60 characters
    - It is stored in the **12 direct block and the 3 indirect block pointers**.
  - Else, **one extra data block** is allocated



Short summary

- Hard link
  - **A directory entry** pointing to an existing file
  - They point to the same inode (no new file content)
  - A file with two pathname
  - Remove file == unlink (link count - 1)
  - Examples: dot/dot dot

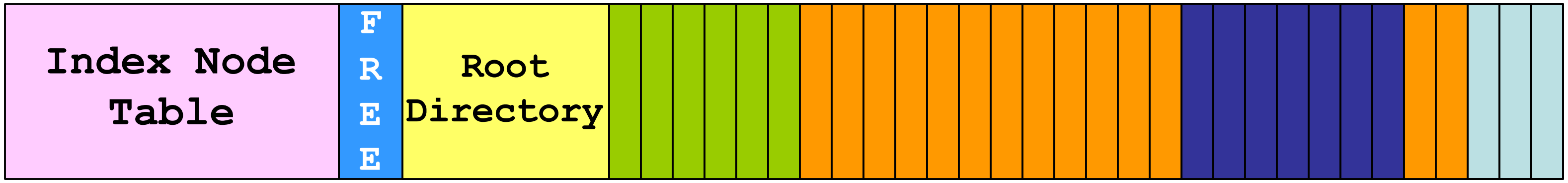
- Symbolic link
  - **A file with a new inode**
  - Stores the target pathname
  - Shortcuts

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- **Buffer cache**
- Journaling
- VFS

File system performance

- Recall the read/write process
  - Directory traversal
  - Reading inode
  - Data blocks



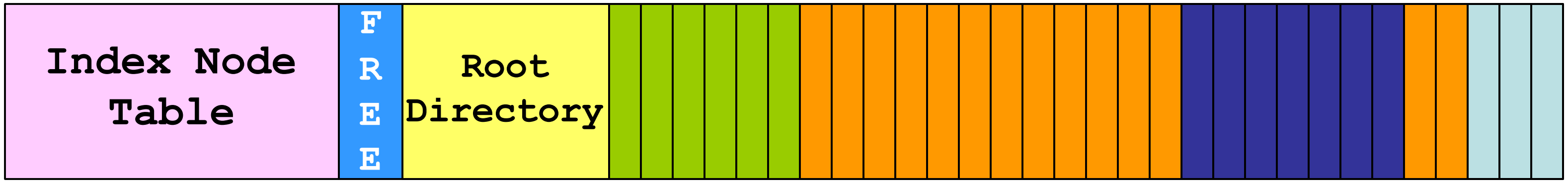
How to improve file system performance?

Kernel Buffer Cache

- Kernel Buffer Cache
  - The kernel will keep a set of copies of the read/written data blocks.
  - The space that stores those blocks are called the **buffer cache**.
  - It is used for **reducing the time** in accessing those blocks in the near future
- Why effective?
  - **Principle of locality**

Kernel Buffer Cache

- What need to be cached?
  - Data blocks, directory file, inode?
  - All of them can benefit from caching



Kernel Buffer Cache

- Three types of buffer caches!

Page Cache	It buffers the data blocks of an opened file.
Directory entry (dcache) cache	Directory entry is stored in the kernel.
Inode cache	The content of an inode is stored in the kernel temporary.

Remember, those cached data is stored in the kernel even though the **corresponding file is closed**!

By the way, the cache is managed under the LRU algorithm.

Kernel Buffer Cache

Read/write mode with kernel buffer cache

Mode	Description
Reading mode	When a process reads a file, the data will be cached automatically. E.g., Readahead system call

Ways	Descriptions
System call	**`ssize_t readahead(int fd, off64_t offset, size_t count);`** A **blocking system call** that stores requested range of data into the kernel page caches Later **`read()`** calls over the range **will not block**.

Readahead

- How does it work?
  - When a file reading operation is requesting for **Block x**, there is a chance that **Block x+1** will also be needed.
  - Such a chance depends on:
    - The file reading mode: sequential access or random access.
    - The file reading history: whether the process **prefers** reading sequentially or not.
  - If such a chance is high, then reading a series of continuous blocks will **reduce the number of disk accesses**. Why?
    - Because the disk head is not always stopped at your desired locations.
    - Because a mechanical disk is good at reading sequential data.
    - How about SSD?

Kernel Buffer Cache

Read/write mode with kernel buffer cache

How about write?

Mode	Description
Write-through mode	Both the **on-disk** and the **cached** copies **update together**. E.g., The write() system call will not return until the on-disk copy is written.
Write-back mode	When a piece of data is going to be written to a file, the **cached copy is updated first**. The update of the on-disk copy is delayed. **On-demand writing dirty blocks back.** Command: **`sync`** System calls: **`sync()`**, **`fsync()`**

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- Buffer cache
- **Journaling**
- VFS

File System Consistency

- Think about caching…tradeoff?
  - System inconsistency exists
    - Power failure, pressing reset button accidentally; etc.
- Disk only provides
  - atomic write of one sector at a time
- A write may require modifying several sectors
  - How to atomically update file system from one consistent state to another?

The **file system journal** is the current, state-of-the-art practice.

Example: Journaling File System

You write down all the tasks assigned to you into a log book.

Task list:
1) Buy boss a DC.
2) Pick up boss' friend.
3) Drive his friend back to his home.
4) Buy boss a coffee when I return.

Your boss orders you to do a set of tasks!

Example: Journaling File System

Task list:

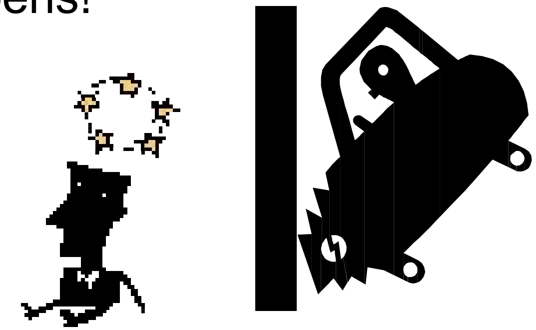
1) ~~Buy boss a DC.~~
2) Pick up boss' friend.
3) Drive his friend back to his home.
4) Buy boss a coffee when I return.

You cross out a task when it is completed.

Example: Journaling File System

Unfortunately, a car accident happens!

You lost all your memory!!



The log book comes in handy!

Your boss sends your colleague to finish your job. But, he doesn't know about your progress.

Worse, your boss has forgotten what are the tasks given to you!

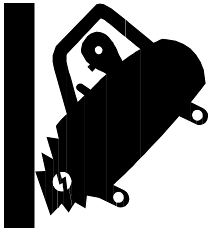
Example: Journaling File System

User Program

FS operations invoked by the user program

Task list:
1) Buy boss a DC.
2) Pick up boss' friend.
3) Drive his friend back to his home.
4) Buy boss a coffee when I return.

OS



System crash!
All memory lost!

File system recovery tool

The journal!

What is journal?

- A journal is the log book for the file system.
  - It is kept inside the file system, i.e., inside the disk.

FS Data	Journal	Data blocks

a new item

- In database: **Write-ahead logging**
- In file systems: **Journaling**
  - Applications: Linux ext3 and ext4, Windows NTFS

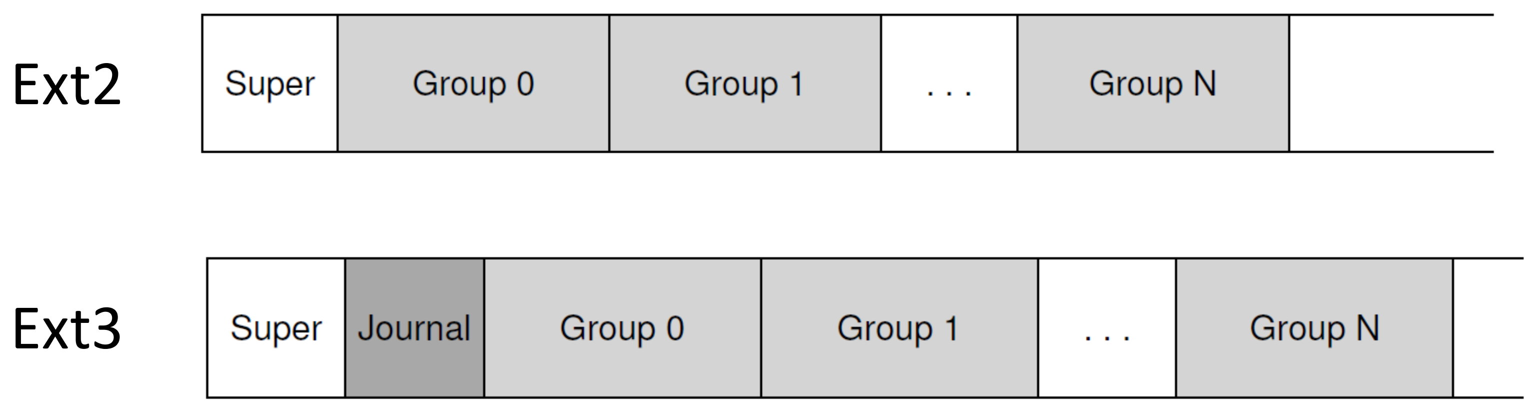
Basic idea: when updating the disk, before overwriting the structures in place, first write down **a little note** describing what you are about to do

What is journal?

- In order to make use of the journal:
  - A set of file system operations becomes an atomic **transaction.**
    - Either all operations are completed successfully, or
    - no operation is completed.
  - A transaction marks all the changes that **will be done** to the FS.
  - Every transaction is written to the journal.

Journaling in Linux ext3

- How does Linux ext3 incorporate the journaling?
  - Most of on-disk structures are identical to Linux ext2
  - The new key structure is the journal itself
  - It occupies some small amount of space within the partition or on another device



Data Journaling

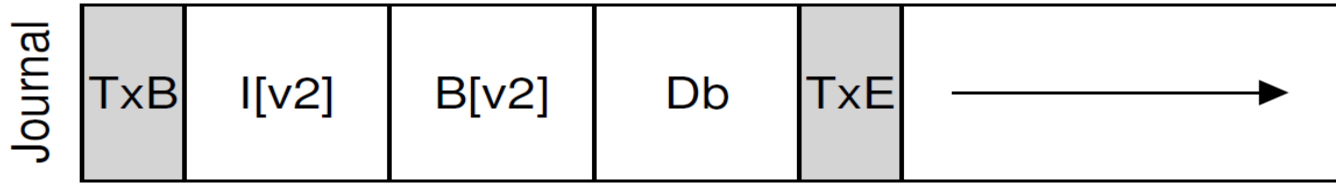
- How to do journaling?

- **Task**: update inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk
  - Metadata + data

- **Strategy: Data journaling**
  - Write all data (metadata+data) to journal
    - Before writing them to their final disk locations, we first write them to log (a.k.a. journal)
  - An available mode with the Linux ext3 file system

Data Journaling

- **Journal layout**:



  - TxB: Transaction begin block
    - It contains some kind of **transaction identifier (TID)**
  - TxE: Transaction end block
    - Marker of the end of this transaction
    - It also contain the TID
- **Checkpoint**
  - Overwrite the old structures in the file system after the transaction being safely on disk

Data Journaling

- Operation sequence:
  - **Journal write**
    - Write the transaction to log and wait for these writes to complete
    - TxB, all pending data, metadata updates, TxE
  - **Checkpoint**
    - Write the pending metadata and data updates to their final locations
- Any problem with this flow?
  - What if crash occurs during the writes to journal

Data Journaling

- We need to write the set of blocks (TxB, I[v2], B[v2], Db, TxE)
  - **Issue one block at a time**
    - It is slow because of waiting for each to complete
  - **Issue all blocks at once**
    - Five writes -> a single sequential write: Faster way
    - However, it is unsafe…
      - The disk internally may perform scheduling and complete small pieces of the big write **in any order**

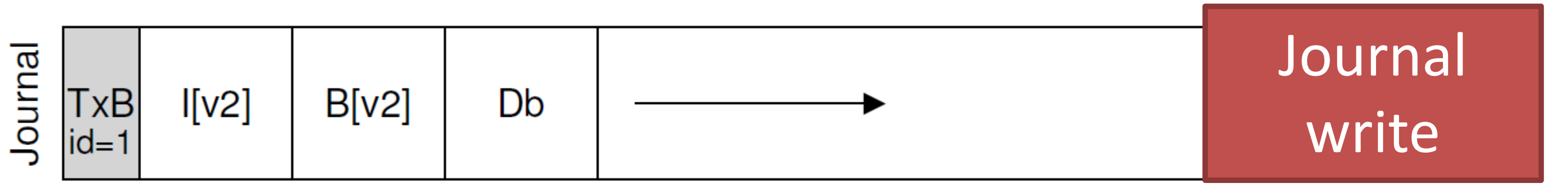
Data Journaling

- **Issue all blocks at once**
  - **Suppose**: disk internally
    - (1) writes TxB, I[v2], B[v2], TxE and later
    - (2) writes Db
  - When crash occurs during the writes to journal
    - If the disk loses power between (1) and (2)

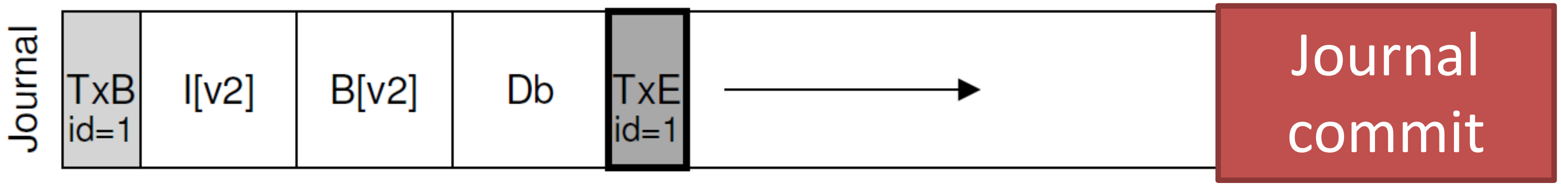
Problem: Transaction looks like a valid transaction, but
the file system can't look at the fourth block and know it is wrong

Data Journaling

- How to solve this problem?
  - Issue transactional write in two steps
    - **First step**: writes all blocks **except the TxE block** to journal



    - **Second step**: file system issues the write of the TxE



Make sure the write of TxE is atomic

Data Journaling

- Operation sequence:
  - **Journal write**
    - Write the contents of the transaction (including TxB, metadata, and data)
  - **Journal commit**
    - metadata, and data (including TxE)
  - **Checkpoint**
    - Write the contents of the update to their on-disk locations

The write order must be guaranteed

Data Journaling

- How to do recovery?
  - Case 1: crash happens **before journal commit**

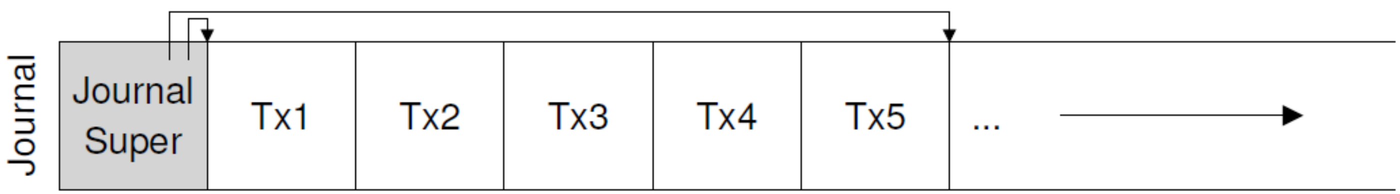
Easy! Skip the pending update

  - Case 2: crash happens **after journal commit, but before checkpoint**

Replay transactions in order. Called **redo logging**

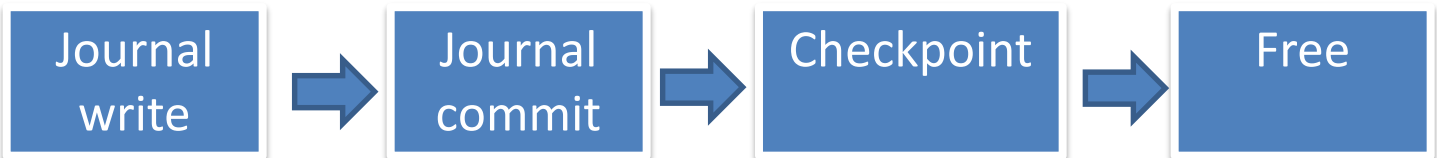
Data Journaling

- The log is of finite size
  - What problems may arise if it is full?
    - Long time to replay
    - Unable to append new transactions
- Manage as a **circular log**
  - **Free** space after checkpointing



Data Journaling

- Write sequence



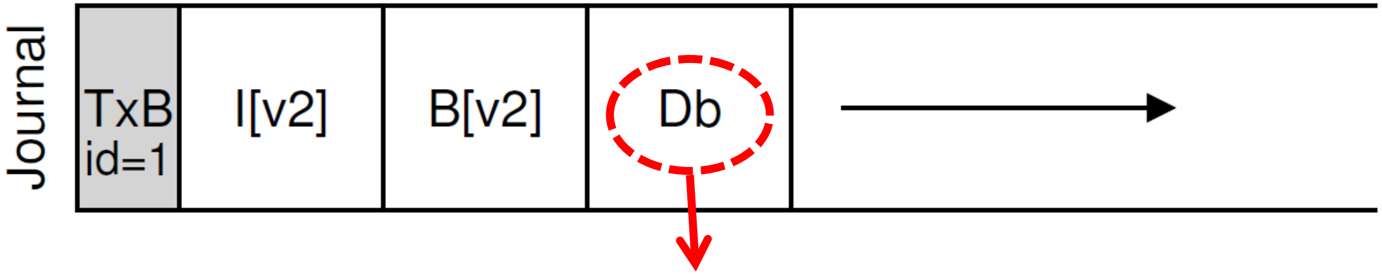
- Data Journaling Timeline

Journal				File System	
TxB	Contents (metadata)	Contents (data)	TxE	Metadata	Data
issue	**issue**	**issue**			
complete					
	complete				
		complete			
			issue		
			complete		
				issue	**issue**
					complete
				complete	

Metadata Journaling

- Any problem with data journaling?
  - Write every Db to disk **twice**
    - Commit to log (journal file)
    - Checkpoint to on-disk location

- How to avoid writing twice?
  - **Metadata journaling**: Logging metadata only



This data is not written to journal

Metadata Journaling

- **Write-back mode**: no order restriction (data/journal)
  - How about data is written to disk after journal commit?
    - File system is consistent (from the perspective of metadata)
    - Metadata points to **garbage data**

- **Ordered mode**
  - Data is written to file system before journal commit
  - Rule:
    - ***Write the pointed-to object before the object that points to it***
    - Core of crash consistency
  - Widely deployed by Ext3, NTFS, etc.

Metadata Journaling

- Write sequence

Data write → Journal metadata write → Journal commit → Checkpoint metadata → Free

The two writes can be issued in parallel

	Journal		File System	
TxB	Contents (metadata)	TxE	Metadata	Data
issue	**issue**			**issue**
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Summary on journal

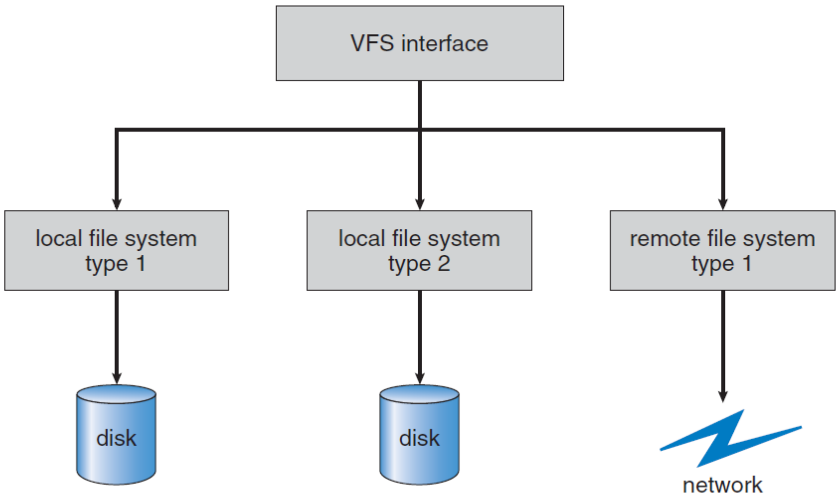
- Working principle:
  - All the changes to the FS are **written to the journal first**, including:
    - the changes in the metadata, i.e., information other than the file content. E.g., the inodes, the directory entries, etc.
    - the file data (depends on data journaling/metadata journaling)
  - Then, the system call returns to the user process.
  - Meanwhile, **the entries in the journal are replayed** and the changes are reflected to the actual file system.

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- Buffer cache
- Journaling
- **VFS**

Virtual File System (VFS)

- Old days: “the” file system
- Nowadays: many fs types and instances co-exist



VFS: an FS abstraction layer

– Transparently and uniformly supports multiple FSes
– A VFS specifies an interface
– A specific FS implements this interface

VFS

- Let’s look into the implementation of **open()**.

```
struct file_operations {
    loff (*llseek)...
    ssize_t (*read)...
    ......
    int (*open) ...
    ......
}
```

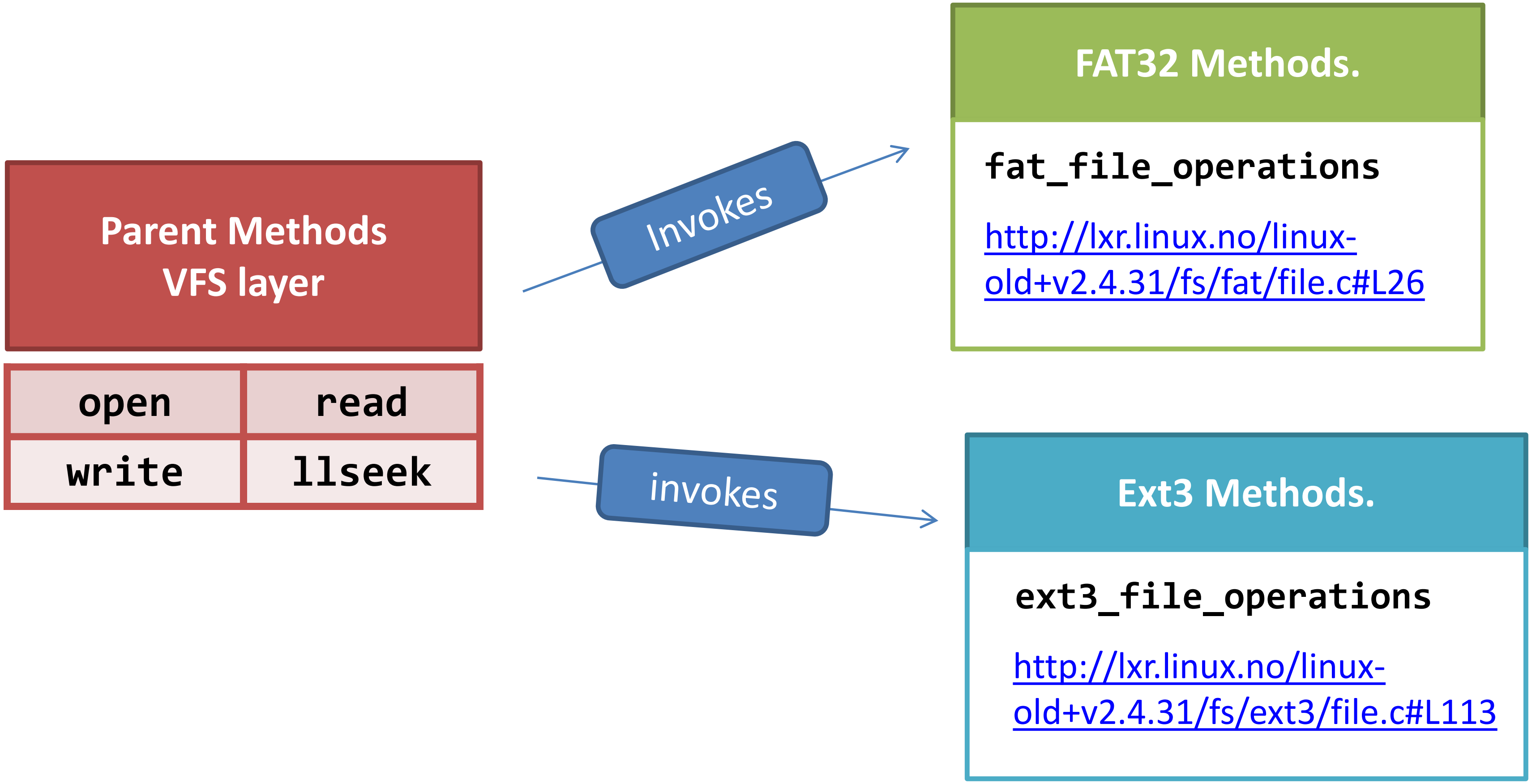
```
struct file
```

```
if (f->f_op && f->f_op->open) {
        error = f->f_op->open(inode,f);
        if (error)
                goto cleanup_all;
}
```

http://lxr.linux.no/linux-old+v2.4.31/fs/open.c

VFS

- For each file system, they have their own set of file operations.



VFS

- So, the beauty in such design is that:
  - The caller, i.e. the VFS layer, doesn't need to care about nor hard-coding which FS you are working on.

```
error = f->f_op->open(inode,f);
```

The only things that require hard-coding are:
- The definition of the file operations.
- The assignment of file operation structures for each FS.

VFS

- A follow-up question is:
  - What if a FS does not support a particular subset of operations?
  - E.g., FAT32 does not need to implement **`chmod()`**!
  - Solution?
    - Simple! Using NULL pointers!
    - When a NULL pointer to a file is detected, returning an error or proceed without any changes.

Summary

- Ext* file systems are the primary FS for Linux
  - They follow the index-node allocation
  - We talked about…
    - Detailed layout (grouping, bitmaps)
    - Inode structure
    - Directory structure
    - Link file (hard link and symbolic link)
    - Kernel buffer cache and readahead
    - Journaling (data journaling, metadata journaling)
    - VFS