

# File and Directory in Practice



Operating Systems  
Wenbo Shen

# Two Key Abstractions

---

- File
  - A linear array of bytes, with each you can read/write
  - Has a low-level name (user does not know) - **inode number**
  - Usually os does not know the exact type of the file
- Directory
  - Has a low-level name
  - Its content is quite specific: contains a list of user-readable name to low-level name. Like ("foo", inode number 10)
  - Each entry either points to file or other directory

# Two Key Abstractions

---

- File
  - External name (visible to user) must be symbolic
    - In hierarchical file system, unique external names are given as pathnames (path from root to the file)
  - Internal names (low-level names): i-node in UNIX
    - Stores information about file system object: file, directory, socket, ...
    - Index into array of file descriptors/headers for a volume
- Directory: translation from external to internal name

# File System Interface

---

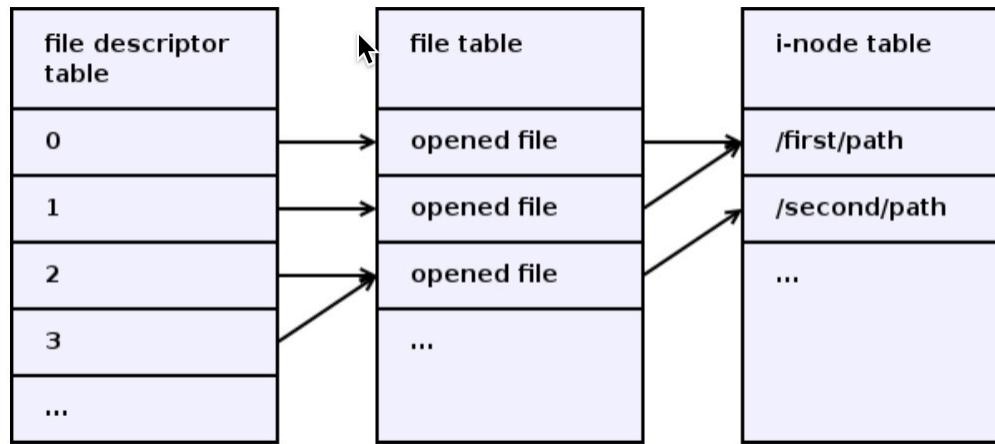
- Create file

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

- *O\_CREAT*: create the file if not exists (not *O\_CREATE*)
- *O\_WRONLY*: can only write to the file
- *O\_TRUNC*: if the file exists, truncate it to zero bytes

The return value of **open()** is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (**read(2)**, **write(2)**, **lseek(2)**, **fcntl(2)**, etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

# File Descriptor



A file descriptor is an index in the per-process file descriptor table (in the left of the picture). Each file descriptor table entry contains a reference to a file object, stored in the file table (in the middle of the picture). Each file object contains a reference to an i-node, stored in the i-node table (in the right of the picture).

A file descriptor is just a number that is used to refer a file object from the user space. A file object represents an opened file. It contains things like current read/write offset, non-blocking flag and another non-persistent state. An i-node represents a filesystem object. It contains things like file meta-information (e.g. owner and permissions) and references to data blocks.

File descriptors created by several `open()` calls for the same file path point to different file objects, but these file objects point to the same i-node. Duplicated file descriptors created by `dup2()` or `fork()` point to the same file object.

A BSD lock and an Open file description lock is associated with a file object, while a POSIX record lock is associated with an [i-node, pid] pair. We'll discuss it below.

# The cat example

---

```
os@os:~/temp$ echo hello > foo
os@os:~/temp$ cat foo
hello
os@os:~/temp$
```

```
-----  
os@os:~/temp$ strace cat foo
```

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(130, 1), ...}) = 0
open("foo", O_RDONLY)                 = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=6, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f703ea95000
read(3, "hello\n", 131072)           = 6
write(1, "hello\n", 6hello
)                                = 6
read(3, "", 131072)                 = 0
munmap(0x7f703ea95000, 139264)      = 0
close(3)                            = 0
close(1)                            = 0
close(2)                            = 0
exit_group(0)                      = ?
+++ exited with 0 +++
```

# Write and fsync

---

- `write()`:
  - please write this data to persistent storage, at some point in the future. The file system, for performance reasons, will buffer such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the `write(s)` will actually be issued to the storage device
- `Fsync()`: forces all dirty (not yet written) data to disk

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

# Getting Information About Files

- Information is kept in **inode**

```
struct stat {  
    dev_t      st_dev;      /* ID of device containing file */  
    ino_t      st_ino;      /* inode number */  
    mode_t     st_mode;     /* protection */  
    nlink_t    st_nlink;    /* number of hard links */  
    uid_t      st_uid;      /* user ID of owner */  
    gid_t      st_gid;      /* group ID of owner */  
    dev_t      st_rdev;     /* device ID (if special file) */  
    off_t      st_size;     /* total size, in bytes */  
    blksize_t  st_blksize;  /* blocksize for filesystem I/O */  
    blkcnt_t   st_blocks;   /* number of blocks allocated */  
    time_t     st_atime;    /* time of last access */  
    time_t     st_mtime;    /* time of last modification */  
    time_t     st_ctime;    /* time of last status change */  
};
```

```
os@os:~/temp$ stat foo  
File: 'foo'  
  Size: 6          Blocks: 8          IO Block: 4096  regular file  
Device: 801h/2049d  Inode: 1328649    Links: 1  
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)  Gid: ( 1000/      os)  
Access: 2018-12-19 00:24:02.448286431 +0800  
Modify: 2018-12-19 00:24:01.316296543 +0800  
Change: 2018-12-19 00:24:01.316296543 +0800  
 Birth: -
```

# Removing Files

---

```
os@os:~/temp$ strace rm foo
```

- Why we just “remove” or “delete” the file, but using “unlinkat”?

```
newfstatat(AT_FDCWD, "foo", {st_mode=S_IFREG|0664, st_size=6, ...}, AT_SYMLINK_NOFOLLOW) = 0
faccessat(AT_FDCWD, "foo", W_OK)          = 0
unlinkat(AT_FDCWD, "foo", 0)           = 0
lseek(0, 0, SEEK_CUR)                     = -1 ESPIPE (Illegal seek)
close(0)                                = 0
```

# Links in FS

---

- Link
  - A file may be known by more than one name in one or more directories. Such multiple names are known as links.
  - The two kinds of links are also known as hard links and soft links.
- Hard link
  - a hard link is a **directory entry** that associates with a file
  - The file name “.” in a directory is a hard link to the directory itself
  - The file name “..” is a hard link to the parent directory
- Soft link (a.k.a., symbolic link or symlink)
  - A symbolic link is a **file** containing the path name of another file
  - Soft links, unlike hard links, may point to directories and may cross file-system boundaries

# Hard Links

---

- `link()` system call: system call takes two arguments, an old pathname and a new one
  - another way to refer to the same file

```
os@os:~/temp/foo$ echo hello > file
os@os:~/temp/foo$ cat file
hello
os@os:~/temp/foo$ ln file file2
os@os:~/temp/foo$ cat file2
hello
os@os:~/temp/foo$ ls -i file file2
1328650 file 1328650 file2
os@os:~/temp/foo$
```

- same inode number
- Even an empty directory has two entries

```
os@os:~/temp/foo$ ls -al
total 8
drwxrwxr-x 2 os os 4096 Dec 20 23:56 .
drwxrwxr-x 3 os os 4096 Dec 20 23:56 ..
```

```
os@os:~/temp/foo$ ls -al
total 8
drwxrwxr-x 2 os os 4096 Dec 20 23:56 .
drwxrwxr-x 3 os os 4096 Dec 20 23:56 ..
```

# Why removing a file calls `unlink`

---

- Create a file
  - making a structure (the inode) that will track virtually all relevant information about the file
  - **linking** a human-readable name to that file (or inode), and putting that link into a directory
- To remove a file, we **unlink** it

`unlink()` deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

# Reference count

```
os@os:~/temp/foo$ rm file
os@os:~/temp/foo$ cat file2
hello
os@os:~/temp/foo$
```

- **rm:** unlink the file, and check the reference count of the inode

```
os@os:~/temp/foo$ echo hello >file
os@os:~/temp/foo$ stat file
  File: 'file'
  Size: 6          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 1328650    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)  Gid: ( 1000/      os)
Access: 2018-12-21 00:09:14.763058468 +0800
Modify: 2018-12-21 00:09:14.763058468 +0800
Change: 2018-12-21 00:09:14.763058468 +0800
 Birth: -
os@os:~/temp/foo$ ln file file2
os@os:~/temp/foo$ stat file2
  File: 'file2'
  Size: 6          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 1328650    Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)  Gid: ( 1000/      os)
Access: 2018-12-21 00:09:14.763058468 +0800
Modify: 2018-12-21 00:09:14.763058468 +0800
Change: 2018-12-21 00:09:24.247463616 +0800
 Birth: -
os@os:~/temp/foo$ ln file2 file3
os@os:~/temp/foo$ stat file
  File: 'file'
  Size: 6          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 1328650    Links: 3
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)  Gid: ( 1000/      os)
Access: 2018-12-21 00:09:14.763058468 +0800
Modify: 2018-12-21 00:09:14.763058468 +0800
Change: 2018-12-21 00:09:33.975880487 +0800
 Birth: -
```

```
os@os:~/temp/foo$ rm file
os@os:~/temp/foo$ stat file2
  File: 'file2'
  Size: 6          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 1328650    Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)  Gid: ( 1000/      os)
Access: 2018-12-21 00:09:14.763058468 +0800
Modify: 2018-12-21 00:09:14.763058468 +0800
Change: 2018-12-21 00:09:40.776172622 +0800
 Birth: -
os@os:~/temp/foo$ rm file2
os@os:~/temp/foo$ stat file3
  File: 'file3'
  Size: 6          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 1328650    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/      os)  Gid: ( 1000/      os)
Access: 2018-12-21 00:09:14.763058468 +0800
Modify: 2018-12-21 00:09:14.763058468 +0800
Change: 2018-12-21 00:09:47.092444491 +0800
 Birth: -
```

# Symbolic links

```
os@os:~/temp/foo$ echo hello > file  
os@os:~/temp/foo$ ln -s file file2
```

```
os@os:~/temp/foo$ ls -l  
total 4  
-rw-rw-r-- 1 os os 6 Dec 21 00:11 file  
lrwxrwxrwx 1 os os 4 Dec 21 00:12 file2 -> file  
os@os:~/temp/foo$ stat file  
  File: 'file'  
  Size: 6          Blocks: 8          IO Block: 4096   regular file  
Device: 801h/2049d  Inode: 1328650      Links: 1  
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    os)  Gid: ( 1000/    os)  
Access: 2018-12-21 00:11:54.636605978 +0800  
Modify: 2018-12-21 00:11:54.636605978 +0800  
Change: 2018-12-21 00:11:54.636605978 +0800  
 Birth: -  
os@os:~/temp/foo$ stat file2  
  File: 'file2' -> 'file'  
  Size: 4          Blocks: 0          IO Block: 4096   symbolic link  
Device: 801h/2049d  Inode: 1328653      Links: 1  
Access: (0777/lrwxrwxrwx)  Uid: ( 1000/    os)  Gid: ( 1000/    os)  
Access: 2018-12-21 00:12:02.300152148 +0800  
Modify: 2018-12-21 00:12:00.960229971 +0800  
Change: 2018-12-21 00:12:00.960229971 +0800  
 Birth: -
```

```
os@os:~/temp/foo$ rm file  
os@os:~/temp/foo$ cat file2  
cat: file2: No such file or directory  
os@os:~/temp/foo$ ls -l  
total 0  
lrwxrwxrwx 1 os os 4 Dec 21 00:12 file2 -> file  
os@os:~/temp/foo$ _
```

# Two different aspects of FS

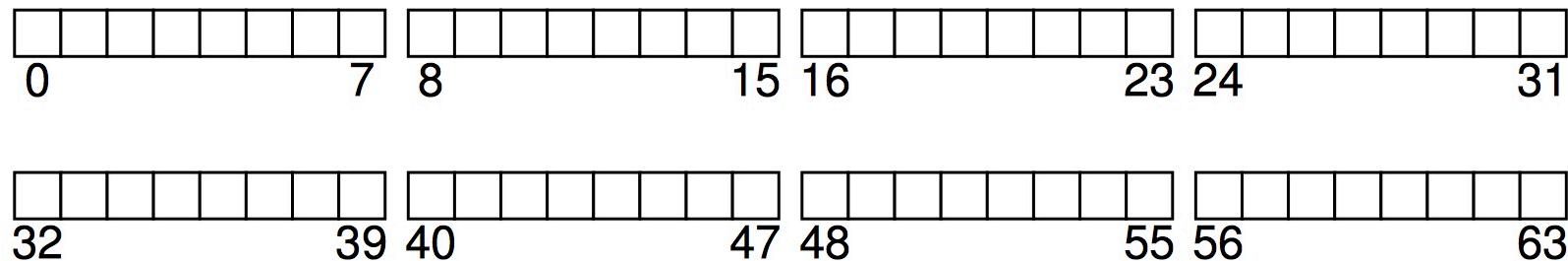
---

- (on-disk) Structure about FS
- Access methods
  - how does the FS maps the calls (open/read/write) onto its structures

# An Example

---

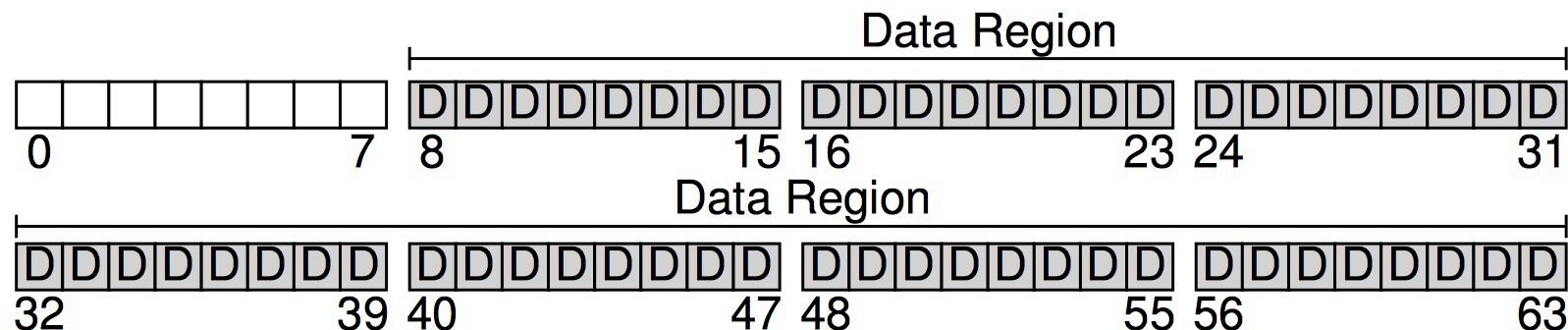
- Suppose we have a serial of blocks
  - Block size: 4k
  - 64 blocks



# Data Region

---

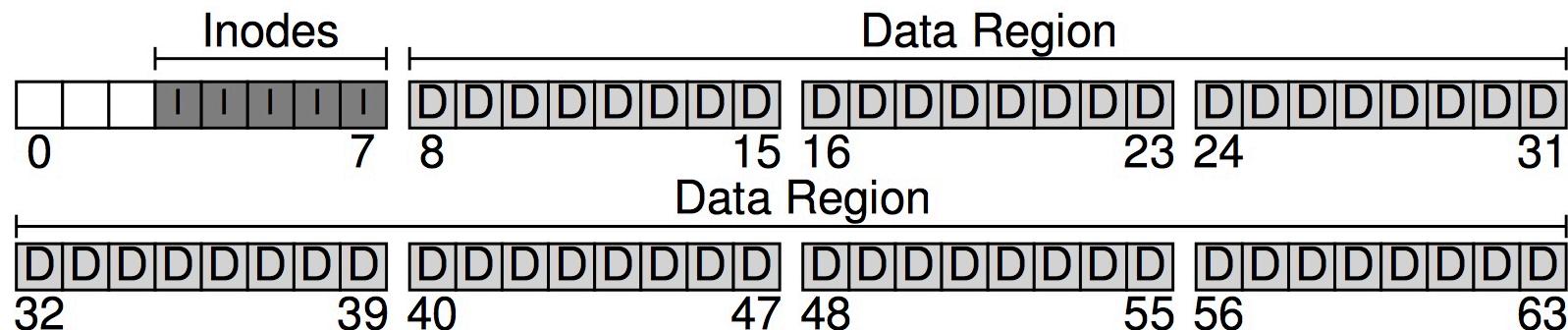
- We reserve some blocks for data
  - 56 of 64 blocks



# Inodes

---

- Inode table: contains inodes
- 5 of 64 blocks are reserved for inodes
  - Suppose inodes are 256 bytes, 4 kb block can hold 16 inodes, then 5 blocks -> 80 inodes -> 80 files (directories)

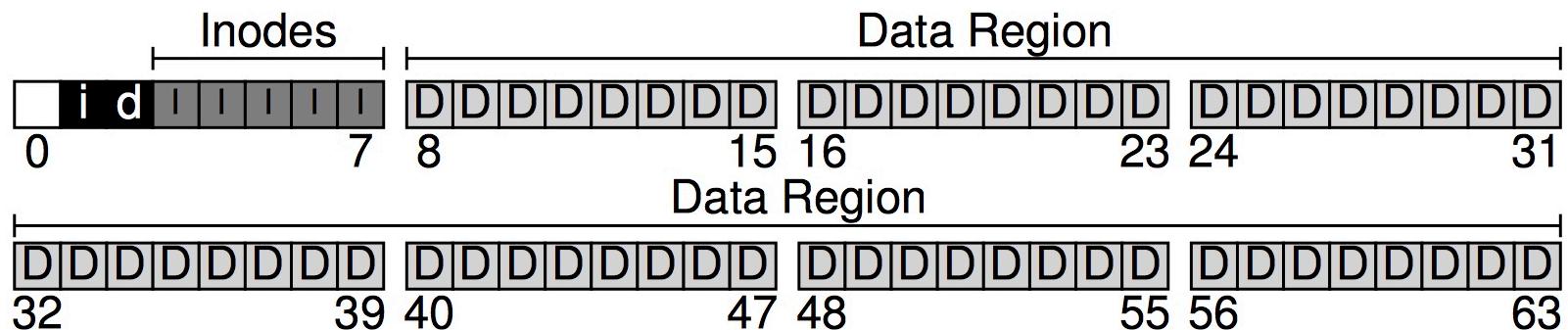


D: data block  
I: inode table

# Bitmap

---

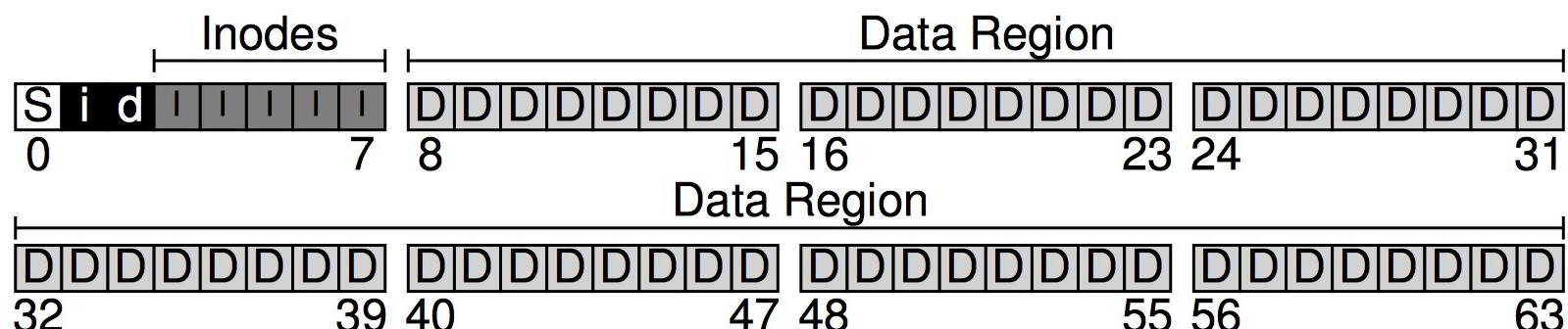
- Suppose we use bitmap to manage the free space
  - One bitmap for free inodes
  - One bitmap for free data region



D: Data block  
I: inode  
i: inode bitmap  
d: data region bitmap

# Superblock

- Superblock
    - Contains information about this file system: how many inodes/data blocks, where the inode table begins, where the data region begins, and a **magic number**



D: Data block  
I: inode  
i: inode bitmap  
d: data region bitmap  
S: superblock

# Inode address calculation

- Suppose inodes are 256 bytes, 4 kb block can hold 16 inodes, then 5 blocks  $\rightarrow$  80 inodes  $\rightarrow$  80 files (directories)
- Each inode is identified by a number(inumber)
- To read inode number No. 32
  - $32 * \text{sizeof(inode)} = 8k$
  - Address:  $8k + 4k(\text{super block}) + 8k(\text{BITMAP}) = 20K$

~~The Inode Table (Closeup)~~

				iblock 0	iblock 1	iblock 2	iblock 3	iblock 4
Super	i-bmap	d-bmap		0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 64 65 66 67	4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 68 69 70 71	8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 72 73 74 75	12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63 76 77 78 79	
0KB	4KB	8KB	12KB	16KB	20KB	24KB	28KB	32KB

# Linux inode

---

```
struct inode {
    umode_t          i_mode;
    unsigned short   i_opflags;
    kuid_t           i_uid;
    kgid_t           i_gid;
    unsigned int     i_flags;

#ifndef CONFIG_FS_POSIX_ACL
    struct posix_acl *i_acl;
    struct posix_acl *i_default_acl;
#endif

    const struct inode_operations *i_op;
    struct super_block    *i_sb;
    struct address_space   *i_mapping;

#ifndef CONFIG_SECURITY
    void              *i_security;
#endif
}
```

# Linux inode

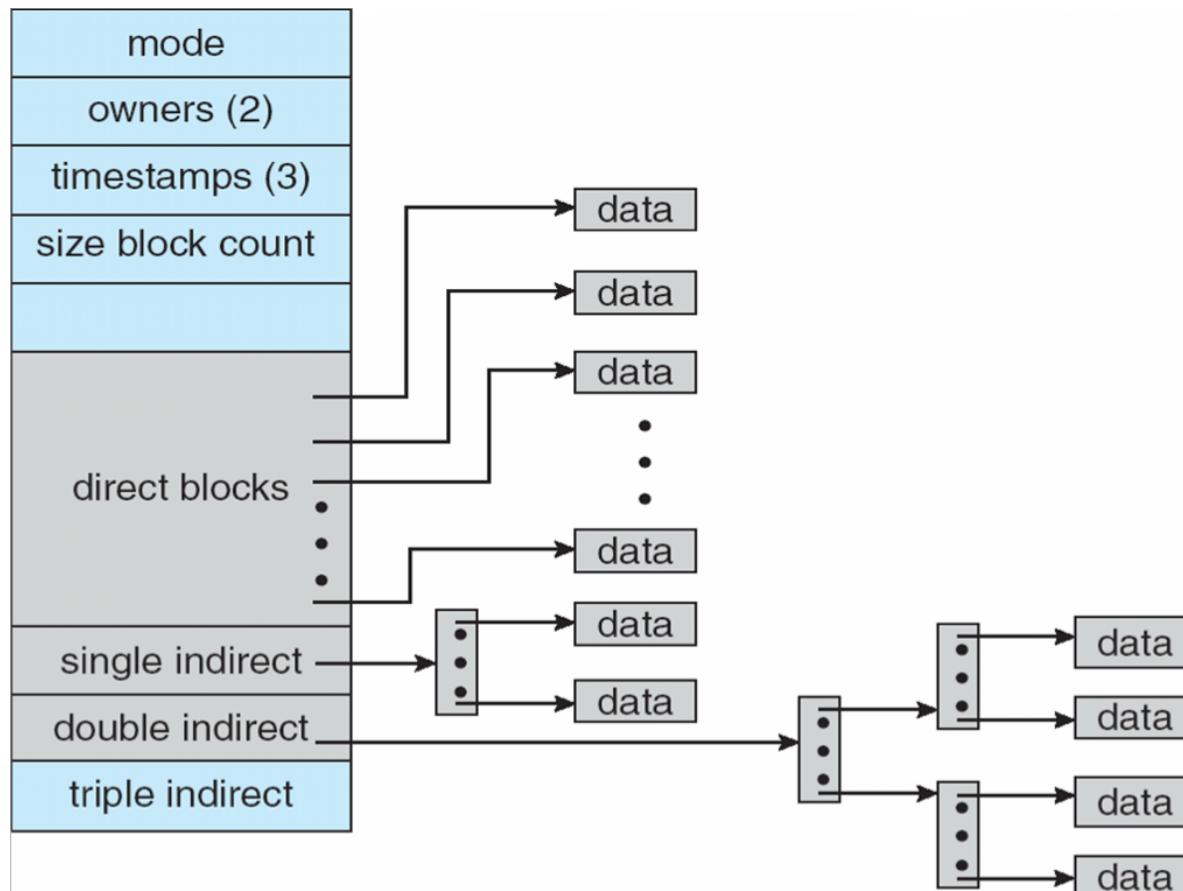
---

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: Simplified Ext2 Inode

# Multi-Level Index

- To support bigger file, we need multi-level index for the nodes



# Directory Organization

---

- Suppose a dir (inode number 5) has 3 files: foo, bar, foobar
  - inode number, name;
  - strlen: length of the name
  - reclen: length of the name plus **left over space**
    - For reuse the entry purpose

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

# Free Space Management

---

- Bit map
- Some OS will use the pre-allocation policy
  - For instance, when a file is created, a sequence of blocks (say 8) will be allocated
    - This can guarantee that the file on the disk is contiguous

# Read /foo/bar

---

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			<b>read</b>			<b>read</b>				
				<b>read</b>				<b>read</b>		
					<b>read</b>					
read()					<b>read</b>			<b>read</b>		
						<b>write</b>				
read()						<b>read</b>			<b>read</b>	
							<b>write</b>			
read()							<b>read</b>			<b>read</b>
								<b>read</b>		
									<b>read</b>	

What about the system-wide/per-process open file table?

# Write to Disk: /foo/bar

---

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		<b>read write</b>	<b>read</b>	<b>read</b>		<b>read</b>	<b>read</b>			
					<b>read write</b>		<b>read</b>			
				<b>write</b>			<b>write</b>			
write()		<b>read write</b>			<b>read</b>					
					<b>write</b>					
write()		<b>read write</b>			<b>read</b>					
					<b>write</b>					
write()		<b>read write</b>			<b>read</b>					
					<b>write</b>					

# Caching and Buffering

---

- Without caching, each file open would require two reads for each level of the directory
  - One for the inode, and one for data
- Early system allocate a **fixed-size cache** to hold popular blocks
- Modern systems use a **unified page cache** for both virtual memory pages and file system pages
- Write buffering: does not write to disk immediately, instead sync to disk for like 5 - 30 seconds
- Database: direct IO with raw data

# Takeaway

---

- File interfaces
  - External name
  - Internal name (low-level name): inode
- Directory
  - Translate from external name to internal name
- Hard link vs soft link
- On-disk layout of FS