

Lecture 12.2

File Systems on Disk

blocks, inodes, the page cache, journaling

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./ext2cat disk.img /hello.txt (real)
block size 1024, 1024 inodes per group
hello.txt in inode 2 -> inode 17
hello from ext2
# 7 pread() calls, no mount, no root
```

What you will learn today



Describe how ext2 lays out a disk: superblock, block groups, inode table, data blocks.



Find a file's data from its path: directory entries, inodes, direct and indirect blocks, ext4 extents.



Explain the page cache, dirty pages, writeback and fsync, with measurements.



Explain why crashes break file systems and how a journal fixes it.

Today's plan (60 min)

0-5

Recap of 12.1

5-12

Disks and blocks

12-30

ext2 on disk, ext4

30-42

Page cache, writeback

42-54

Crashes, journal

54-60

Lab and checkpoint

Lab 12.2: ext2cat.c, a file system reader.

Quick recap of Lecture 12.1

Q1

After dup, do the two fds share the file offset?

Yes: two fds, one struct file, one f_pos. Two open() calls give two offsets.

Q2

Where is a file's name kept?

In a directory entry: (name → inode number). The inode has no name; that allows hard links.

Q3

How does one read() work for ext4, pipes and /proc?

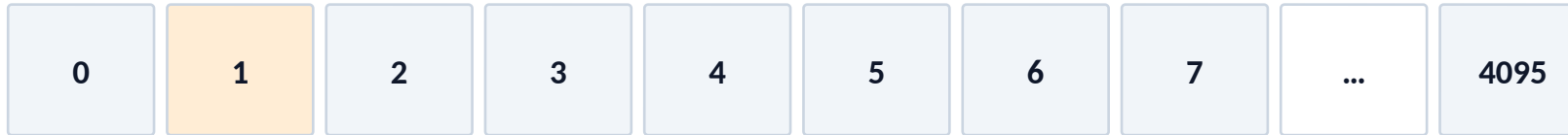
vfs_read calls file->f_op->read_iter (or ->read): each file system fills that table.

PART 1

Disks and blocks

Below the VFS, a disk is only a long array of numbered blocks.

A disk is an array of blocks; a file system is a data structure in it



disk.img: 4096 blocks of 1024 bytes. Block n starts at byte $n \times 1024$.

```
● ● ● C link: "read block n"
pread(fd, buf, 1024,
      n * 1024);
```

level	unit	size (our machines)
disk hardware	sector	512 bytes (/sys/block/vda/queue/logical_block_size)
file system	block	4096 (ext4 image) or 1024 (lab ext2 image)
page cache	page	4096 (Lecture 11)
a file	bytes 0 ... size - 1	any length; the last block is partly used

SAME AS C

Just a struct in an array

Today: C structs at known offsets in one big byte array. Lab 12.2 reads them with `pread`.

Analogy: an exercise book. The pages are numbered (blocks). The file system adds a contents page (superblock), index cards (inodes) and a list of chapters by name (directories). Lose the contents page and the pages are still there, but nobody can find anything.

No root needed: a file system can live inside an ordinary file (a disk image).

Book: OSTEP, "File System Implementation"; Love, LKD Ch. 14

Below the page cache: the block layer (real trace)

● ● ● QEMU guest: cold dd of 1 MiB; events block_rq_issue / _complete

```
dd block_rq_issue: 253,0 RA 16384 () 33144 + 32
<idle> d.h2. block_rq_complete: 253,0 RA () 33144 + 32
dd block_rq_issue: 253,0 RA 32768 () 33176 + 64
dd block_rq_issue: 253,0 RA 65536 () 33240 + 128
<idle> d.h2. block_rq_complete: 253,0 RA () 33176 + 64
<idle> dNh2. block_rq_complete: 253,0 RA () 33240 + 128
dd block_rq_issue: 253,0 RA 131072 () 33368 + 256
dd d.H1. block_rq_complete: 253,0 RA () 33368 + 256
dd block_rq_issue: 253,0 RA 131072 () 33624 + 256
... and so on, 128 KiB each
```

The I/O scheduler sits between the file system and the driver: it merges and orders requests. Our guest: none [mq-deadline] (cat /sys/block/vda/queue/scheduler). For hard disks, order matters: moving the head costs milliseconds.

253,0: the device: major 253 = virtio disk vda

RA: read-ahead: the kernel reads more than dd asked for

33144 + 32: start sector + number of 512-byte sectors (16 KiB)

16 → 32 → 64 → 128 KiB: the read-ahead window doubles while the reads stay sequential

d.h2.: "h" = this line ran inside a hardware interrupt: the disk said "done" (Lecture 13)

Read-ahead is why sequential reads are fast: the next pages are already coming while you process this one.

Book: Love, LKD Ch. 14 (the block I/O layer)

PART 2

ext2 on disk

The superblock, block groups, inodes, blocks and directories: byte by byte.

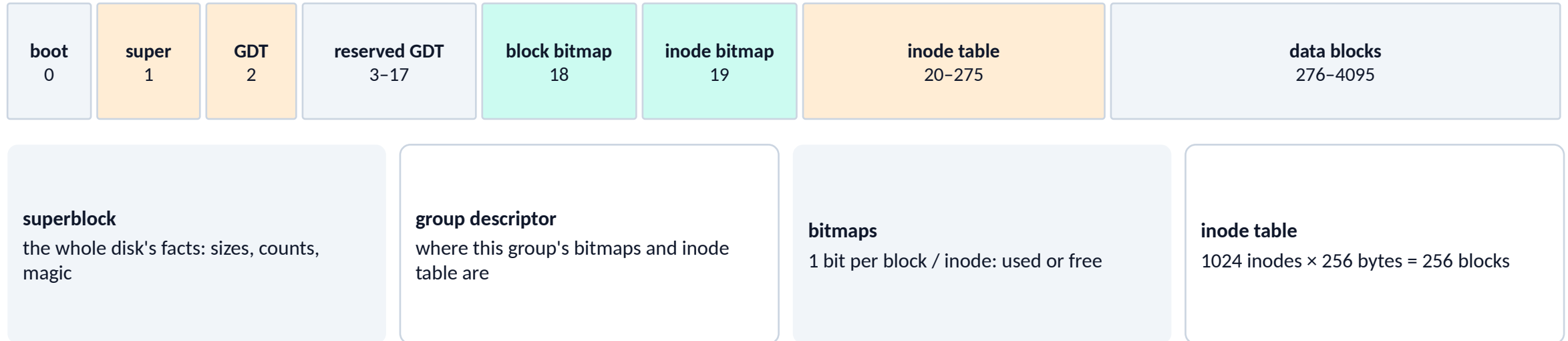
Our lab disk: ext2 in a 4 MiB file

mkimage.sh (core lines; no root needed)

```
$ echo "hello from ext2" > files/hello.txt
$ seq 1 30000 > files/docs/numbers.txt
$ ln -s notes.txt files/docs/readme
$ mkfs.ext2 -q -b 1024 -d files disk.img 4M
```

dumpe2fs disk.img (real, shortened)

```
Filesystem magic number: 0xEF53
Block size: 1024          Block count: 4096
Inode size: 256           Inode count: 1024
Group 0: (Blocks 1-4095)
  Primary superblock at 1, Group descriptors at 2-2
  Block bitmap at 18, Inode bitmap at 19
  Inode table at 20-275
```



Book: OSTEP, "File System Implementation"; Bovet & Cesati Ch. 18 (ext2)

Step 1 and 2: superblock (byte 1024) and group descriptor

```
fs/ext2/ext2.h: struct ext2_super_block (real)
410 struct ext2_super_block {
411     __le32 s_inodes_count;
412     __le32 s_blocks_count;
413     ...
416     __le32 s_first_data_block;
417     __le32 s_log_block_size; /* 1024 << this */
419     __le32 s_blocks_per_group;
421     __le32 s_inodes_per_group;
422     ...
426     __le16 s_magic; /* 0xEF53 */
427     ...
451     __le16 s_inode_size;
```

DIFFERENT FROM C

le32, not int

Exact sizes and byte order are part of the format: the lab uses `uint32_t` and `uint16_t`. x86 is little-endian too: no conversion.

```
od -t x1 -j 1080 -N 2 disk.img (real)
0001080 53 ef
# byte 1024 + 56: s_magic = 0xEF53, little-endian
```

```
od -t x1 -j 2048 -N 16 disk.img (real)
0002048 12 00 00 00 13 00 00 00 14 00 00 00 ...
bg_block_bitmap = 0x12 = 18
bg_inode_bitmap = 0x13 = 19
bg_inode_table = 0x14 = 20
```

Group descriptors are 32 bytes each, in the block after the superblock: block (`s_first_data_block + 1`) = block 2 here.

Book: Bovet & Cesati §18.2 (ext2 disk data structures)

Step 3: find inode n, and read what it says

1 Inodes count from 1: $\text{group} = (n - 1) / 1024$, $\text{index} = (n - 1) \% 1024$

2 hello.txt is inode 17: group 0, index 16

3 $\text{byte} = \text{inode_table} \times 1024 + \text{index} \times 256 = 20 \times 1024 + 16 \times 256 = 24576$

```
fs/ext2/ext2.h: struct ext2_inode (real)
```

```
290 struct ext2_inode {
291     __le16 i_mode;        /* File mode */
292     __le16 i_uid;
293     __le32 i_size;        /* Size in bytes */
294-7   __le32 i_atime, i_ctime, i_mtime, i_dtime;
299     __le16 i_links_count;
300     __le32 i_blocks;      /* 512-byte units */
313     __le32 i_block[EXT2_N_BLOCKS]; /* 15 */
```

```
od -t x1 -j 24576 -N 48 disk.img (real)
```

```
a4 81 00 00 10 00 00 00 c2 b6 cf 6a c2 b6 cf 6a
c2 b6 cf 6a 00 00 00 00 00 00 01 00 02 00 00 00
00 00 00 00 00 00 00 00 22 03 00 00 00 00 00 00
```

field	bytes	value
i_mode	a4 81	0x81a4 = 0100644: regular, rw-r--r--
i_size	10 00 00 00	16 bytes
i_links_count	01 00	1 name
i_blocks	02 00 00 00	2 × 512 = one 1 KiB block
i_block[0]	22 03 00 00	0x322 = block 802

Book: Bovet & Cesati §18.2.4 (inodes)

i_block[15]: 12 direct pointers, then pointers to pointers

i_block[0-11] direct: data blocks 0-11

i_block[12] single indirect: a block of 256 block numbers

i_block[13] double indirect: 256×256 blocks

i_block[14] triple indirect: 256^3 blocks

Why this shape? Most files are small: 12 direct pointers reach them with no extra reads. Big files pay one extra block read per level. With 1 KiB blocks a block holds $1024 / 4 = 256$ block numbers.

```
debugfs -R "stat ..." disk.img (real)
```

```
# hello.txt, 16 bytes  
(0):802  
# numbers.txt, 168894 bytes = 165 blocks  
(0-11):636-647, (IND):648, (12-164):649-801  
# big.txt, 348894 bytes = 341 blocks  
(0-11):291-302, (IND):303, (12-267):304-559,  
(DIND):560, (IND):561, (268-340):562-634
```

SAME AS C

Pointers to pointers

A single indirect block is an array of block numbers; a double indirect block is an array of those: pointers to pointers, as in C, but with block numbers instead of addresses.

Book: OSTEP, "File System Implementation" (the multi-level index)

Step 4: a directory block is a chain of entries

fs/ext2/ext2.h (real)

```
591 struct ext2_dir_entry_2 {  
592     __le32 inode;    /* Inode number */  
593     __le16 rec_len;  /* Directory entry length */  
594     __u8  name_len; /* Name length */  
595     __u8  file_type;  
596     char  name[];    /* no '\0' */
```

root directory = inode 2 → block 276: od (real)

```
02 00 00 00 0c 00 01 02 2e 00 00 00  
02 00 00 00 0c 00 02 02 2e 2e 00 00  
0b 00 00 00 14 00 0a 02 6c 6f 73 74 2b 66 6f 75 6e 64 00 00  
0c 00 00 00 0c 00 04 02 64 6f 63 73  
11 00 00 00 c8 03 09 01 68 65 6c 6c 6f 2e 74 78 74 ...
```

inode	rec_len	name_len	type	name
2	12	1	2 dir	.
2	12	2	2 dir	..
11	20	10	2 dir	lost+found
12	12	4	2 dir	docs
17	968	9	1 file	hello.txt

1 Start at offset 0; read the 8-byte header

2 Compare name_len bytes of name; **no '\0'** to rely on

3 Jump: offset += rec_len. The last entry's rec_len (968) fills the block

Book: Bovet & Cesati §18.2.6 (directories)

Watch ext2cat walk from a path to the bytes

ext2cat: from a path to bytes, by hand

SUPERGROUPINODEDIRDATA

disk.img: 4 MiB = 4096 blocks of 1 KiB (not to scale)

boot	super	GDT	reserved	bitmaps	inode table	/ dir	other files	hello	free
0	1	2	3-17	18, 19	20-275	276	277-801	802	803-4095

```
pread(fd, buf, 1024, 276 * 1024)  block 276
```

inode	rec_len	name
2	12	.
2	12	..
11	20	lost+found
12	12	docs
17	968	hello.txt ← found (968: the rest of the block)

```
what ext2cat knows      pread calls: 4
```

block size 1024
inodes/group 1024
inode size 256
inode table at block 20
/ = inode 2 → block 276
hello.txt = inode 17

Step 4: a directory's data is a list of entries: inode, rec_len (jump to the next one), name_len, type, name. Walk with rec_len and compare names: "hello.txt" is inode 17.

Watch for

1. The **magic** 53 ef at byte 1080.
2. The inode table starts at **block 20**.
3. Root = **inode 2**.
4. **rec_len** jumps through the directory.
5. **7 pread calls** in all.

Click the video to play or pause. About 1 minute, no sound.

ext4: extents instead of block lists

fs/ext4/ext4_extents.h (real)

```
63 struct ext4_extent {
64     __le32 ee_block;    /* first logical block */
65     __le16 ee_len;      /* number of blocks */
66     __le16 ee_start_hi; /* physical block, high */
67     __le32 ee_start_lo; /* physical block, low */
68 };
```

debugfs on the guest's ext4 image (real)

```
# big64.bin, 67108864 bytes, 4 KiB blocks
EXTENTS:
(0-16383):4143-20526
# hello.txt, 20 bytes
EXTENTS:
(0):20527
```

64 MiB file, 4 KiB blocks	ext2 block lists	ext4 extents
file data blocks	16 384	16 384
metadata to find them	17 indirect blocks (64 KiB)	1 extent: 12 bytes, inside the inode
reading block 10 000	inode → DIND → IND → data	inode → data

Extent = (start, length). Like describing a street by "houses 4143 to 20526" instead of listing 16 384 house numbers. Works when the allocator keeps files contiguous; ext4 also has a journal, checksums and 64-bit block numbers.

Book: OSTEP, "File System Implementation" (extents)

PART 3

The page cache and writeback

Reads come from RAM when they can; writes wait in RAM until the kernel writes them back.

Reads: the second time comes from RAM

```
QEMU guest: readtime /mnt/big64.bin (real)
$ echo 3 > /proc/sys/vm/drop_caches # root: empty the caches
Cached: 7644 kB
read 1: 64 MiB in 287.5 ms = 223 MiB/s
read 2: 64 MiB in 40.6 ms = 1577 MiB/s
Cached: 73164 kB # + 64 MiB
```

What it is: all free RAM is used to keep file pages: the "buff/cache" column of free -m. It is given back when programs need memory.

Shared: read(), write() and mmap of the same file use the same pages (Lecture 11: file-backed page faults).

```
test machine: ./readtime big.bin drop (real)
# drop = posix_fadvise(DONTNEED): no root needed
read 1: 256 MiB in 90.0 ms = 2843 MiB/s
read 2: 256 MiB in 35.8 ms = 7154 MiB/s
```

Your lab PC: a hard disk reads about 100–200 MiB/s and seeks take ms: the cold/warm gap is far bigger than on our SSD-backed machines.

Book: Love, LKD Ch. 16 (the page cache)

Writes: write() returns long before the disk is written

```
QEMU guest: writer /mnt/w.bin 32 (real)
```

```
write(): 32 MiB in 141.9 ms
```

```
t=0 s: Dirty: 32760 kB
```

```
t=10 s: Dirty: 20500 kB
```

```
t=20 s: Dirty: 24 kB
```

```
t=30 s: Dirty: 24 kB
```

```
t=40 s: Dirty: 0 kB
```

```
mm/page-writeback.c (real) and /proc/sys/vm
```

```
74 static int dirty_background_ratio = 10;
```

```
91 static int vm_dirty_ratio = 20;
```

```
102 dirty_writeback_interval = 5 * 100; /* cs */
```

```
109 dirty_expire_interval = 30 * 100; /* cs */
```

```
$ cat /proc/sys/vm/dirty_expire_centisecs
```

```
3000
```

```
$ cat /proc/sys/vm/dirty_writeback_centisecs
```

```
500
```

- 1 write() copies into page-cache pages and marks them **dirty**; it returns at once (32 MiB in 0.14 s).
- 2 Flusher threads wake every **5 s** and write pages dirty for more than **30 s**, or sooner when dirty memory passes the **background** limit.
- 3 Our trace said `reason=background`: in a 512 MiB guest, 32 MiB was already too much dirty data. On a big PC, small writes wait about 30 s.

Book: Love, LKD Ch. 16 (page writeback)

fsync: "wait until it is really on the disk"

```
./writer w.bin 256 [fsync] (test machine; real)

write(): 256 MiB in 110.0 ms
# again, with fsync:
write(): 256 MiB in 57.7 ms
fsync(): 209.6 ms
```

```
the path in Linux 6.8 (real lines)

fs/sync.c 218 SYSCALL_DEFINE1(fsync, unsigned int, fd)
fs/sync.c 206 static int do_fsync(...)
fs/sync.c 180 int vfs_fsync_range(...)
                → file->f_op->fsync
fs/ext4/file.c 944 .fsync = ext4_sync_file,
```

```
QEMU guest: writer /mnt/w2.bin 32 fsync (real)

write(): 32 MiB in 133.7 ms
fsync(): 98.4 ms Dirty: 0 kB
```

DIFFERENT FROM C

fflush is not fsync

fflush(fp) moves the stdio buffer into the kernel (page cache). Only fsync(fileno(fp)) waits for the disk.

Three layers of buffering: your stdio buffer (fflush) → the page cache (fsync) → the disk's own cache (fsync also sends a flush command to the disk). Databases call fsync at every commit: that is the price of durability.

Book: Kerrisk, TLPI §13.3 (fsync); Love, LKD Ch. 16

PART 4

Crashes and journaling

Power can fail between any two disk writes. The journal makes updates all-or-nothing.

Why a crash breaks a file system

Appending one block to a file needs **three** separate disk writes:

data block
the new bytes

inode
new size, new pointer

block bitmap
mark the block used

only these reached the disk	result after the crash
data	harmless: the new data is just lost
inode	the file points to a block that may hold garbage, and the bitmap says it is free
bitmap	a block marked used that nobody owns: a space leak
inode + bitmap	file system consistent, but the file shows old garbage
inode + data	the bitmap disagrees: the block can be given to another file

Old fix: fsck. After a crash, scan the WHOLE disk: every inode, every block, every directory, and repair what disagrees. It works, but on a big disk it takes minutes to hours, at every boot after a power cut.

Analogy: a shopkeeper's ledger. Before he moves goods, he writes "moving 3 bags from A to B" in the ledger. If he is interrupted, the ledger tells him what to finish. That ledger is the **journal**.

Book: OSTEP, "Crash Consistency: FSCK and Journaling"

Journaling: write it down first, then do it

1 **Journal write:** copy the changed metadata blocks (inode, bitmap, directory) into the journal

2 **Commit:** write a commit block. The transaction now counts, even if we crash

3 **Checkpoint:** write the same blocks to their real places in the file system

4 **Free:** mark the transaction done; the journal space is reused

5 **After a crash:** replay committed transactions; ignore uncommitted ones

ext4 mode (mount -o data=...)	what goes to the journal
journal	metadata AND data: safest, every byte written twice
ordered (default)	metadata only; data is written BEFORE its metadata commits
writeback	metadata only, no ordering: fastest, old data may appear

● ● ● real: header and kernel facts

```
Journal inode: 8   Total journal size: 16M
include/linux/jbd2.h 48:
#define JBD2_DEFAULT_MAX_COMMIT_AGE 5
/proc/fs/jbd2/vda-8/info:
    36449us average transaction commit time
```

Book: OSTEP, "Crash Consistency" (journaling); Love, LKD Ch. 13–16

Experiment: pull the plug on our guest (real)

● ● ● A: crash at once (sysrq b = reset now)

```
$ writer /mnt/kept.bin 1 fsync
fsync(): 39.0 ms
$ echo "written, never synced" > /mnt/lost.txt
$ echo b > /proc/sysrq-trigger
```

● ● ● B: write, wait 7 s, crash

```
$ echo "written, never synced" > /mnt/late.txt
$ sleep 7; echo b > /proc/sysrq-trigger
```

file	fsync?	after the crash
kept.bin (A)	yes	safe: replayed from the journal
lost.txt (A)	no, crash at once	gone: never committed
late.txt (B)	no, 7 s passed	survived: a commit ran first

● ● ● on the host, with the disk images (real)

```
$ dumpe2fs -h dA.img | grep features
... needs_recovery ...
$ debugfs -R "ls -l /" dA.img # before replay
(no kept.bin, no lost.txt)
$ e2fsck -fy dA.img
dA.img: recovering journal
Free blocks count wrong ... Fix? yes
$ debugfs -R "ls -l /" dA.img # after
21 100644 1048576 kept.bin
$ ... the same for dB.img:
21 100644 22 late.txt
$ debugfs -R "cat /late.txt" dB.img
written, never synced
```

How to save a file safely: write, fsync, rename

● ● ● safesave.c (core lines)

```
fd = open(".settings.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC, 0644);
write(fd, text, strlen(text));
fsync(fd); /* 1. new data is on disk */
close(fd);
rename(".settings.txt.tmp", "settings.txt"); /* 2. atomic */
d = open(".", O_RDONLY | O_DIRECTORY);
fsync(d); /* 3. the rename is on disk */
```

● ● ● strace ./safesave (real, shortened)

```
fsync(3) = 0   rename("./.settings.txt.tmp", "./settings.txt") = 0   fsync(3) = 0
```

Why not just open("settings.txt", O_TRUNC)?

A crash after the truncate and before the data is written leaves an empty file: the old settings are gone too.

Why rename?

rename replaces the name in one step: the directory points to the old inode or to the new one, never to half a file.

Why fsync the directory?

the rename is a change to the directory; without it, a crash can bring back the old name.

Book: Kerrisk, TLPI §13.3, §18.4 (rename)

LAB 12.2

ext2cat.c: a file system reader

superblock → group descriptor → inode → directory → data, with pread.

Lab 12.2 tasks, step by step

1 `./mkimage.sh` (or use the `disk.img` in the zip); look at it with `dumpe2fs` and `debugfs -R "ls -l /"`

2 **TODO 1:** read the superblock at byte 1024; check `0xEF53`; `bs = 1024 << s_log_block_size`

3 **TODO 2:** `read_inode`: group and index, the group descriptor, then the inode

4 **TODO 3:** walk a directory block with `rec_len`; compare names with `memcmp`

5 **TODO 4:** print `i_size` bytes from the direct blocks. Test `/hello.txt`, `/docs/notes.txt`

Check every step with the tools: `debugfs -R "stat /hello.txt" disk.img` prints the inode you should find; `od -A d -t x1 -j BYTE -N 64 disk.img` shows the raw bytes.

● ● ● reference, real

```
$ ./ext2cat disk.img /docs
block size 1024, 1024 inodes per group, ...
docs in inode 2 -> inode 12
inode 12 rec_len 12 .
inode 2 rec_len 12 ..
inode 13 rec_len 16 big.txt
inode 14 rec_len 20 notes.txt
inode 15 rec_len 20 numbers.txt
inode 16 rec_len 944 readme
$ ./ext2cat disk.img /docs/notes.txt
Lab 12.2 notes
line two
$ ./ext2cat disk.img /docs/numbers.txt
block 12 needs an indirect block: Homework 12.2
```

Checkpoint: test yourself

- Q1** At which byte is an ext2 superblock, whatever the block size? → **byte 1024 (magic 0xEF53 at byte 1080)**
- Q2** Inode 17, 1024 inodes per group, inode table at block 20, 256-byte inodes, 1 KiB blocks: where? → **group 0, index 16: $20 \times 1024 + 16 \times 256 = 24576$**
- Q3** write() returned. Is the data on the disk? → **no: dirty pages in the page cache; fsync waits**
- Q4** fflush(fp) or fsync(fd): which one survives a power cut? → **fsync (fflush only reaches the page cache)**
- Q5** What does the journal replay after a crash? → **only committed transactions; the rest are ignored**

Key ideas from today



A disk is an array of blocks; ext2 puts a superblock, group descriptors, bitmaps and an inode table in it.



Path → directory entries → inode → i_block (direct, indirect) or ext4 extents → data.



The page cache serves reads from RAM and holds dirty pages; writeback or fsync writes them.



A crash can stop between writes; the journal commits all-or-nothing. Only fsync promises durability.

Homework 12.2 (due before Lecture 13)

1. Indirect blocks

Add `i_block[12]` and `i_block[13]` to your `ext2cat`. Check `/docs/numbers.txt` and `/docs/big.txt` with `cmp`. What is the largest file with 1 KiB blocks?

2. The symlink

`./ext2cat disk.img /docs/readme` fails with "short read at byte 1742980626432". Explain that number (`debugfs -R "stat /docs/readme"` helps), and fix it.

3. Your disk

Run `writer` with 256 MiB, with and without `fsync`, and `readtime` with `drop`, on a lab PC. Compare with our numbers. Why is `write()` so fast?

4. The crash

Use our crash experiment: why did `kept.bin` survive, `lost.txt` vanish and `late.txt` survive? Why does `safesave.c` `fsync` twice?

5. Directory order

`dirent` on `ext4` lists names in a strange order; `ext2cat` lists our root in creation order. Find the reason in `fs/ext4/dir.c` (`is_dx_dir`).

Next lecture and reading

NEXT: LECTURE 13

Interrupts and exceptions

How the CPU stops what it is doing when a device needs attention: the IDT, interrupt handlers, top and bottom halves, softirqs and timers. Today's trace already showed one: `block_rq_complete` ran inside the disk's interrupt ("h").

Main text

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., Ch. 10 today; §8.1 (exceptions) for Lecture 13.

Kernel side

R. Love, **LKD**, Ch. 14 and 16 today; Ch. 7 (interrupts) and Ch. 8 (bottom halves) for Lecture 13.

Free book

Arpaci-Dusseau, **OSTEP**: "File System Implementation" and "Crash Consistency: FSCK and Journaling".

Lab files

`lab12_files.zip`: `ext2cat_starter.c`, `mkimage.sh`, `disk.img`, and the demos (`readtime`, `writer`).