

Lecture 12.1

Files and the VFS
descriptors, inodes, directories

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./fds (real)
fd 3 -> /etc/hostname  ext4
fd 4 -> pipe:[2835]    pipefs
fd 6 -> socket:[2836]  sockfs
fd 8 -> /proc/539/status  proc
# one read(), many kinds of file
```

What you will learn today



Explain the three tables behind a file descriptor: fd table, open-file table, inode.



Describe inodes, directories, hard and symbolic links, and how a path is looked up.



Explain how the VFS lets one `read()` work on ext4, pipes, `/proc` and devices.



Use `stat`, `readdir`, `lstat` and `readlink` from C.

Today's plan (60 min)

0-5

Recap of 11.2

5-12

Files from C

12-24

Three tables

24-37

Inodes, directories

37-52

The VFS

52-60

Lab and checkpoint

Lab 12.1: `myls.c`, your own `ls -li`.

Quick recap of Lecture 11.2

Q1

malloc(200 KiB): which system call does glibc use?

mmap: the block is above the 128 KiB mmap threshold. Small blocks come from the heap (brk).

Q2

free(p) on a 100-byte block: does the RAM go back to the kernel?

No. The chunk goes on a tcache or bin shelf for the next malloc; the break does not move.

Q3

The machine runs out of RAM after overcommit. What happens?

The OOM killer picks the process with the highest badness and sends SIGKILL (exit status 137).

PART 1

A file, from C to the kernel

FILE * in C, an int in the kernel: the file descriptor.

"Everything is a file": what is behind each descriptor?

./fds (real; run in a terminal)

```
fd 0 -> /dev/pts/0          magic 0x1cd1
fd 1 -> /dev/pts/0          magic 0x1cd1
fd 2 -> /dev/pts/0          magic 0x1cd1
fd 3 -> /etc/hostname        magic 0xef53
fd 4 -> pipe:[2835]         magic 0x50495045
fd 5 -> pipe:[2835]         magic 0x50495045
fd 6 -> socket:[2836]       magic 0x534f434b
fd 7 -> socket:[2837]       magic 0x534f434b
fd 8 -> /proc/539/status     magic 0x9fa0
fd 9 -> /dev/null           magic 0x1021994
fd 10 -> anon_inode:[eventfd] magic 0x9041934
```

DIFFERENT FROM C

Not only disk files

In C you used `fopen` for files on disk. In Linux the same `int fd` can name a pipe, a socket, a terminal, or kernel data.

magic	file system	what it is
0xef53	ext4	files on the disk
0x50495045	pipefs	pipes ("PIPE" in ASCII)
0x534f434b	sockfs	sockets ("SOCK")
0x9fa0	proc	kernel data as text
0x1cd1	devpts	terminals
0x1021994	tmpfs	files in RAM (/dev)

How `fds.c` knows: `readlink("/proc/self/fd/N")` gives the name; `fstatfs(N)` gives the magic number of the file system that owns it (`include/uapi/linux/magic.h`).

Book: CS:APP §10.1–10.2; Kerrisk, TLPI Ch. 4

FILE * (stdio) is a buffer on top of an fd

```
stdio.c: 1000 lines of 32 bytes  
for (i = 0; i < 1000; i++) printf("%s", line);  
/* or, with "write": */  
for (i = 0; i < 1000; i++) write(1, line, 32);
```

```
strace -e trace=write ./stdio > out.txt (real)  
write(1, "a line of text, 32 bytes long...\n...", 4096) = 4096  
write(1, "a line of text, 32 bytes long...\n...", 4096) = 4096  
... 8 calls: 32 000 / 4096 = 7.8
```

run (strace -c -e trace=write)	write calls	why
./stdio > out.txt	8	stdout is a file: full buffering, 4096 bytes
./stdio (terminal)	1000	a terminal: line buffering, one per \n
./stdio write > out.txt	1000	no buffer: every call enters the kernel

SAME AS C

Your fopen, fprintf, fgets

are still right. `fileno(fp)` gives the fd under the FILE *. `fclose` = `fflush` + `close`.

DIFFERENT FROM C

The data waits in YOUR process

until the buffer is full, a `\n` reaches a terminal, or `exit`. A crash (or `_exit`) before that loses it; fork copies it (Lecture 10.1).

Book: CS:APP §10.10 (standard I/O); Kerrisk, TLPI Ch. 13

PART 2

Three tables behind every fd

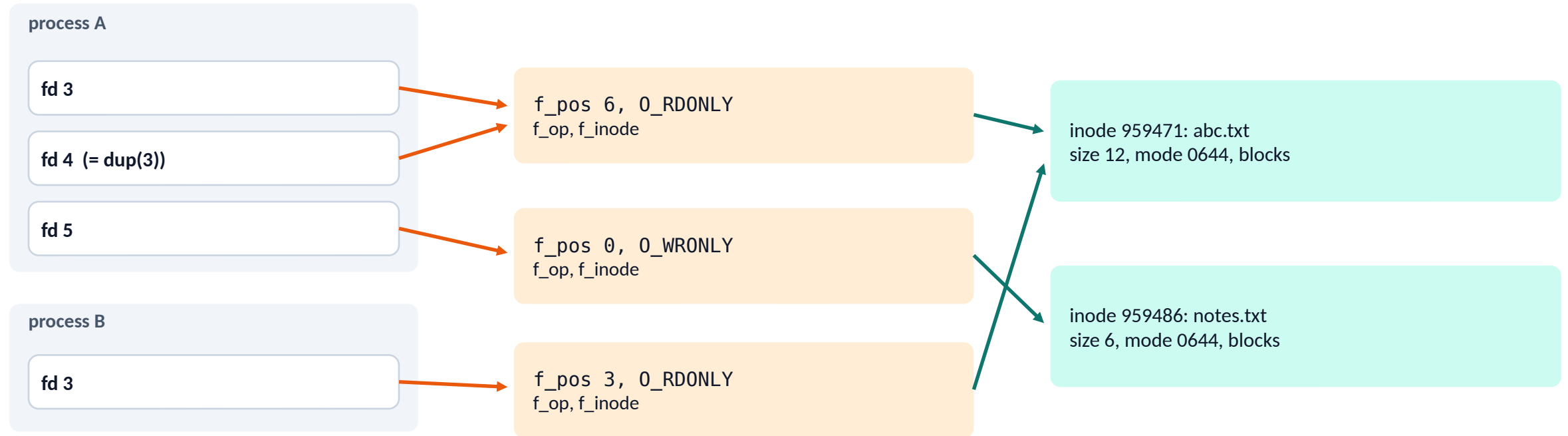
fd table → open-file table → inode: who shares what.

fd → open file → inode: three levels, like a library

fd table
one per process

open-file table
struct file, shared kernel-wide

inode table
one per file on disk



Library analogy: the fd is the number on your borrowing card (only means something to you); the open file is the checkout record with your **bookmark** (f_pos); the inode is the **book** itself. Two readers of one book have two bookmarks.

Book: CS:APP §10.8 (sharing files); Kerrisk, TLPI §5.4

Proof: who shares the offset? (share.c)

```
./share (abc.txt holds "abcdefghijkl"; real)
```

```
1) open() twice: two open-file entries
   a = open()      fd 3 read "abc"  offset now 3
   b = open()      fd 4 read "abc"  offset now 3
2) dup(): one open-file entry, two fds
   a = open()      fd 3 read "abc"  offset now 3
   b = dup(a)      fd 4 read "def"  offset now 6
3) fork(): the child inherits the SAME entry
   child           fd 3 read "abc"  offset now 3
   parent, after child fd 3 read "def" offset now 6
```

Two open()s: two struct files, two offsets: both read "abc".

dup(): two fds, ONE struct file: b continues where a stopped (Lecture 10.1's dup2 pipes).

fork(): the child's fd table is a copy, but its entries point to the parent's struct files: the parent continues after the child.

DIFFERENT FROM C

FILE * copies are different

In C, `FILE *g = f`; copies a pointer to the same FILE: one buffer, one position. Two `fopen()` calls on one file give two FILE objects: two buffers, two positions, like case 1.

Why it matters: a shell does `cmd > out.txt` once, then forks. Every process in `(echo a; echo b) > out.txt` writes at the shared offset, so the lines follow each other instead of overwriting.

Book: CS:APP §10.8–10.9

The same three tables in Linux 6.8

● ● ● include/linux/fdtable.h (real lines)

```
24 #define NR_OPEN_DEFAULT BITS_PER_LONG /* 64 */
27 struct fdtable {
28     unsigned int max_fds;
29     struct file __rcu **fd; /* current fd array */
30     unsigned long *close_on_exec;
31     unsigned long *open_fds; /* a bitmap */
```

● ● ● fs/file.c: alloc_fd (real lines)

```
499 static int alloc_fd(unsigned start, unsigned end, ...)
510     if (fd < files->next_fd)
511         fd = files->next_fd;
514         fd = find_next_fd(fdt, fd); /* lowest free */
524     error = expand_files(files, fd);
```

● ● ● include/linux/fs.h: struct file (real lines)

```
994 struct file {
1008     fmode_t      f_mode;
1009     atomic_long_t f_count; /* how many fds */
1011     loff_t        f_pos;
1012     unsigned int  f_flags; /* O_APPEND, ... */
1016     struct path   f_path;
1017     struct inode   *f_inode;
1018     const struct file_operations *f_op;
```

C link: the fd is just an **index** into struct file *fd[]. open() returns the **lowest free** index: that is why close(1) + open() makes the new file stdout (the old way to redirect).

Book: Love, LKD Ch. 13 (the file object)

The offset can jump: lseek and files with holes

● ● ● sparse.c (core lines)

```
fd = open("sparse.bin", O_WRONLY | O_CREAT | O_TRUNC, 0644);
lseek(fd, 1L << 30, SEEK_SET); /* 1 GiB: only f_pos moves */
write(fd, "x", 1);
fstat(fd, &st);
```

● ● ● ./sparse ; ls -ls ; du -h (real, ext4)

```
st_size  = 1073741825 bytes
st_blocks = 8 (x 512 = 4096 bytes really on disk)
$ ls -ls sparse.bin
4 -rw-r--r-- 1 root root 1073741825 Oct 14 22:56 sparse.bin
$ du -h sparse.bin
4.0K    sparse.bin
```

1 lseek only changes **f_pos** in the struct file: no disk work.

2 The write lands at byte 1 073 741 824: the file's **size** becomes 1 GiB + 1.

3 The 1 GiB gap was never written: ext4 stores **no blocks** for it (a hole). Reads there return zeros.

4 **st_size** is the length; **st_blocks** is the real space, in 512-byte units.

Book: Kerrisk, TLPI §4.7 (file holes)

PART 3

Inodes and directories

A file has a number, not a name. Directories hold the names.

The inode: everything about a file except its name

```
stat proj/notes.txt (real)
File: proj/notes.txt
Size: 6      Blocks: 8      IO Block: 4096  regular file
Device: 254,0    Inode: 959486  Links: 1
Access: (0644/-rw-r--r--)  Uid: (0/root)  Gid: (0/root)
Access: 2026-10-14 22:56:41.156990582 +0600
Modify: 2026-10-14 22:56:41.156990582 +0600
Change: 2026-10-14 22:56:41.156990582 +0600
```

st_mode = 0100644 (octal): 4 bits of type, 3 special bits, then rwx for owner, group, others

1000	000	110	100	100
type (4)	s (3)	owner (3)	group (3)	others (3)
1000 = regular file (S_IFREG)	110 = rw-	100 = r--		

struct stat field	meaning
st_ino	the inode number: the file's real identity
st_mode	type + permissions (S_ISDIR, S_ISLNK, ...)
st_nlink	how many names (directory entries) point here
st_uid, st_gid	owner and group
st_size / st_blocks	length / space used (512-byte units)
st_atime, mtime, ctime	read, content change, inode change

Not in the inode: the name. The name lives in the directory. That is how one file can have two names (hard links) or none at all (deleted but still open).

Book: CS:APP §10.6; Kerrisk, TLPI Ch. 15

A directory is a table of (name → inode number)

```
./dirent proj (readdir; real, ext4)
```

```
inode 959483 type dir src
inode 959482 type dir .
inode 959486 type file notes.txt
inode 959356 type dir ..
inode 959484 type dir docs
inode 959487 type link readme
```

```
ls -lai proj (real)
```

```
959482 drwxr-xr-x 4 root root 4096 .
959356 drwxr-xr-x 4 root root 4096 ..
959484 drwxr-xr-x 2 root root 4096 docs
959486 -rw-r--r-- 1 root root 6 notes.txt
959487 lrwxrwxrwx 1 root root 9 readme -> notes.txt
959483 drwxr-xr-x 2 root root 4096 src
```

Not sorted: readdir gives the order the file system keeps; ext4 uses a hash order. ls sorts the names itself.

"." and ".." are real entries: "." is the directory's own inode, ".." its parent's. That is how cd .. works.

Link count 4: proj has 4 names: "proj" in its parent, its own ".", and ".." in src and docs. Empty directory: 2.

C link: struct dirent { ino_t d_ino; unsigned char d_type; char d_name[256]; }: readdir returns one per call, like reading records from an array of structs.

Book: CS:APP §10.7; Kerrisk, TLPI Ch. 18

Hard links and symbolic links

```
./links.sh (real)
$ echo hello > a
$ ln a b          # a second NAME for the same inode
$ ln -s a c       # a small file holding the path "a"
$ ls -li
959478 -rw-r--r-- 2 root root 6 a
959478 -rw-r--r-- 2 root root 6 b
959479 lrwxrwxrwx 1 root root 1 c -> a
$ rm a
$ ls -li
959478 -rw-r--r-- 1 root root 6 b
959479 lrwxrwxrwx 1 root root 1 c -> a
$ cat b
hello
$ cat c
cat: c: No such file or directory
```

	hard link (ln)	symbolic link (ln -s)
is	another directory entry, same inode	a new inode whose content is a path
link count	goes up (2)	unchanged
original removed	still works: data stays until count 0	dangling: the path leads nowhere
across file systems	no (inode numbers are per disk)	yes
to a directory	not allowed	allowed

SAME AS C

Pointers again

A hard link is a second pointer to the same struct; a symlink is a char * holding a name, looked up again each time it is used.

Book: Kerrisk, TLPI §18.1–18.2

Delete a file that is open: the name goes, the file stays

```
● ● ● unlink.c (core lines)

fd = open("secret.txt", O_RDWR | O_CREAT | O_TRUNC, 0600);
write(fd, "still here\n", 11);
unlink("secret.txt");           /* remove the NAME */
fstat(fd, &st); ... pread(fd, buf, 31, 0);
```

```
● ● ● ./unlinked (real)

after unlink: inode 959474, links 0, size 11
/proc/503/fd/3 -> /home/claude/l12/demo/secret.txt (deleted)
read through fd 3: still here
```

Two counters decide when a file dies: `i_nlink` (names) and the open references (struct files). Both must reach 0.

1 unlink removes the directory entry: **links 0**. Nobody can open it by name any more.

2 The struct file still points to the inode: **reads and writes work**.

3 close (or exit) drops the last reference: **now** the inode and its blocks are freed.

Real uses: programs make temporary files this way (nobody else can find them, and they vanish even after a crash). **Real trap:** `rm` a 500 MiB log that a server still writes, and `df` shows no space freed (Homework 12.1).

Book: Kerrisk, TLPI §18.3 (unlink)

Path lookup: one directory at a time

```
namei -l ../proj/src/main.c (real)

drwxr-xr-x root root /
drwxr-xr-x root root home
drwxr-xr-x claude claude claude
drwxr-xr-x root root l12
drwxr-xr-x root root demo
drwxr-xr-x root root proj
drwxr-xr-x root root src
-rw-r--r-- root root main.c
```

open("/mnt/a/b/c/file.txt")	1st run (cold)	2nd run
mnt	lookup_fast	lookup_fast
a	lookup_slow → ext4_lookup	lookup_fast
b	lookup_slow → ext4_lookup	lookup_fast
c	lookup_slow → ext4_lookup	lookup_fast
file.txt	lookup_open → ext4_lookup	lookup_fast

fttrace in the QEMU guest after dropping the caches (real; fs/namei.c)

The dentry cache (dcache): every name the kernel has looked up stays in RAM as a dentry (name → inode). The 2nd open needs no disk at all: lookup_fast finds every name in the cache (`__d_lookup_rcu`, namei.c 1628). Only a miss calls the file system's own lookup (`i_op->lookup`, line 1693).

- 1 Start at / (or the current directory if the path is relative).
- 2 For each name: need **x** on the directory, then find the name in it.
- The last name gives the inode; open makes a struct file for it.

Book: Love, LKD Ch. 13 (the dentry object)

PART 4

The virtual file system

One set of system calls; many file systems behind function pointers.

The VFS: one interface, many file systems

user programs: open, read, write, stat, readdir, close, mmap ...

VFS generic code: fd → struct file, path lookup + dcache, permission checks, page cache; then call **f_op** / **i_op** of the file

ext4
disk

proc
kernel data

pipefs
pipes

sockfs
sockets

tmpfs
RAM files

devtmpfs
devices

(also ext2, vfat for USB sticks, nfs over the network, sysfs, ...)

mount attaches a file system's root dentry to a directory of the tree: `mount -t ext4 /dev/vda /mnt` in our guest. One tree, many file systems inside it.

VFS object	what it stands for	operations
super_block	one mounted file system	super_operations
inode	one file (size, mode, ...)	inode_operations
dentry	one name in a path	dentry_operations
file	one open file (f_pos)	file_operations

Analogy: a wall socket. Your laptop charger does not care whether the power comes from a power station, solar panels or a generator. The socket is the interface; the VFS is the socket of the kernel.

Book: Love, LKD Ch. 13; Bovet & Cesati Ch. 12

How C does "objects": a table of function pointers

```
● ● ● include/linux/fs.h: struct file_operations (real)

1983 struct file_operations {
1984     struct module *owner;
1985     loff_t (*llseek) (struct file *, loff_t, int);
1986     ssize_t (*read) (struct file *, char __user *, ...);
1987     ssize_t (*write) (struct file *, const char __user *, ...);
1988     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1989     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1993     __poll_t (*poll) (struct file *, ...);
1996     int (*mmap) (struct file *, struct vm_area_struct *);
    ... open, release, fsync, ...
```

```
● ● ● the same idea in first-semester C

struct shape_ops {
    double (*area)(void *self);
};
double circle_area(void *c) { ... }
double square_area(void *s) { ... }
struct shape_ops circle_ops = { .area = circle_area };

/* the caller does not know the shape: */
x = obj->ops->area(obj);
```

SAME AS C

Plain C

Function pointers, structs and designated initializers (`.read_iter = ...`) from your C course. No new language feature.

DIFFERENT FROM C

Objects by hand

C++ or Java would call this a virtual method table. The kernel writes it by hand, one table per kind of file.

vfs_read calls through the pointer; each file system fills the table

fs/read_write.c: vfs_read (real lines)

```
456 ssize_t vfs_read(struct file *file, char __user *buf,
                    size_t count, loff_t *pos)
460     if (!(file->f_mode & FMODE_READ))
461         return -EBADF;
467     ret = rw_verify_area(READ, file, pos, count);
473     if (file->f_op->read)
474         ret = file->f_op->read(file, buf, count, pos);
475     else if (file->f_op->read_iter)
476         ret = new_sync_read(file, buf, count, pos);
477     else
478         ret = -EINVAL;
```

Read it with Lecture 9: `read(3, buf, 256) → ksys_read → vfs_read → file->f_op->read_iter`. Wrong mode: `-EBADF`, and `libc` sets `errno = EBADF`. The VFS checks; the file system does the work.

fs/ext4/file.c (real)

```
931 const struct file_operations ext4_file_operations = {
932     .llseek      = ext4_llseek,
933     .read_iter   = ext4_file_read_iter,
934     .write_iter  = ext4_file_write_iter,
944     .fsync       = ext4_sync_file,
```

fs/pipe.c (real)

```
1239 const struct file_operations pipefifo_fops = {
1242     .read_iter   = pipe_read,
1243     .write_iter  = pipe_write,
```

fs/proc/base.c (real)

```
789 static const struct file_operations proc_single_...
791     .read        = seq_read,
```

Book: Love, LKD Ch. 13 (file operations)

Proof: the same vfs_read, four different paths (ftrace)

● ● ● readone /mnt/hello.txt (ext4, page cached)

```
vfs_read() {  
  rw_verify_area() { ... }  
  ext4_file_read_iter() {  
    generic_file_read_iter() {  
      filemap_read() {  
        filemap_get_pages(); 17.0 us  
        touch_atime();      8.2 us  
      }  
    }  
  }  
} 131.4 us
```

● ● ● echo hello | readone (pipe)

```
vfs_read() {  
  rw_verify_area() { ... }  
  pipe_read() {  
    mutex_lock();  
    anon_pipe_buf_release();  
    mutex_unlock();  
    touch_atime();  
  }  
} 105.3 us
```

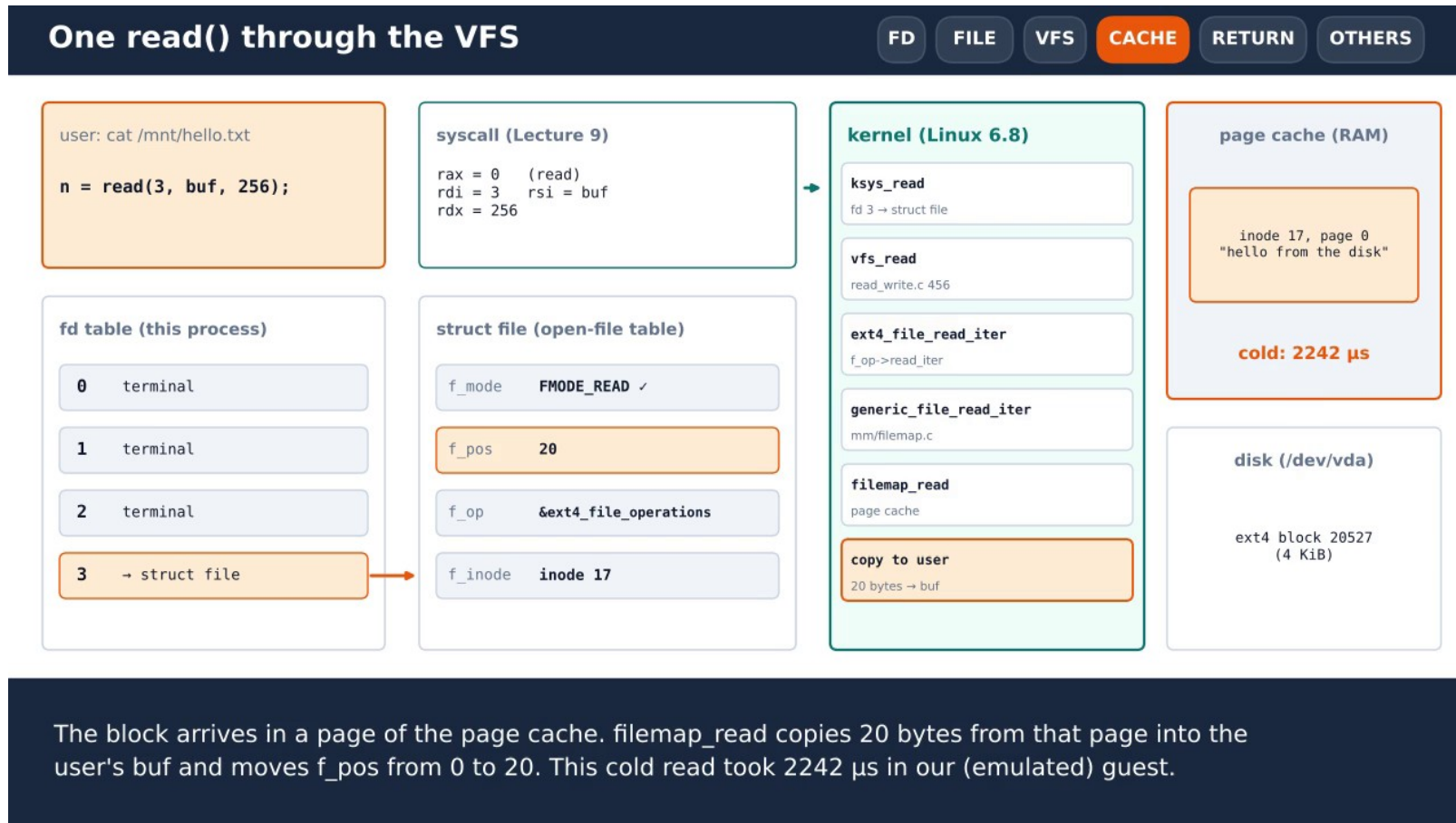
● ● ● readone /proc/self/status (proc)

```
vfs_read() {  
  seq_read() {  
    seq_read_iter(); 3710.7 us  
  }  
} 3785.9 us
```

● ● ● readone /dev/null (a device)

```
vfs_read() {  
  rw_verify_area() { ... }  
  read_null(); 37.4 us  
} 89.5 us
```

Watch one read() go through the VFS



Click the video to play or pause. About 1.3 minutes, no sound.

Watch for

1. fd 3 is only an **index**.
2. struct file holds **f_pos** and **f_op**.
3. vfs_read calls **through f_op**.
4. A miss reads the **disk**, a hit does not.
5. A pipe or /proc: **another f_op**.

open() builds the chain; the page cache makes reads cheap

● ● ● open("/mnt/hello.txt") in the kernel (ftrace, real)

```
do_sys_openat2() {  
  getname()           /* copy the path in */  
  get_unused_fd_flags() {  
    alloc_fd();        /* lowest free fd */  
  }  
  do_filp_open() {  
    path_openat();     1455 us  /* lookup + struct file */  
  }  
  fd_install();        /* fd[3] = file */  
} 1544.7 us
```

read(3, buf, 256) of hello.txt	cold	warm
vfs_read in total (ftrace)	2242 μ s	131 μ s
filemap_read	2165 μ s	50 μ s
disk read?	yes (block 20527)	no

Page cache: file pages kept in RAM, shared by read(), write() and mmap (Lecture 11). Lecture 12.2 measures it, and what happens to written pages before they reach the disk.

1 **open** = path lookup (dentries) + a new struct file (f_op from the inode) + the lowest free fd pointing to it.

2 **read / write** = fd → struct file → f_op → the file system → the page cache → (only on a miss) the disk.

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

LAB 12.1

myls.c: your own ls -li

readdir for the names, lstat for the inodes, readlink for symlinks.

Lab 12.1 tasks, step by step

- 1 **TODO 1** in `mysls_starter.c`: `opendir`, then `readdir` until `NULL`; build `"dir/name"` with `snprintf`
- 2 **TODO 2**: `lstat` the path; print inode, mode, links, owner (`getpwuid`), size, name
- 3 **TODO 3**: `mode_string`: the type letter, then `rxw` three times from the bits `0400` ... `0001`
- 4 **TODO 4**: for a symlink, `readlink` and print `" -> target"` (add the `'\0'` yourself)
- 5 Compare with `ls -lia` on your home directory, `/dev` and `/proc/self/fd`

lstat, not stat: `stat` follows a symlink and describes the target; for `readme -> notes.txt` you would print a regular file and never see the `"l"`.

```
./mysls proj (reference; real)

959483 drwxr-xr-x 2 root 4096 src
959482 drwxr-xr-x 4 root 4096 .
959486 -rw-r--r-- 1 root 6 notes.txt
959356 drwxr-xr-x 4 root 4096 ..
959484 drwxr-xr-x 2 root 4096 docs
959487 lrwxrwxrwx 1 root 9 readme
                        -> notes.txt

$ ls -lia proj
959482 drwxr-xr-x 4 root root 4096 .
959356 drwxr-xr-x 4 root root 4096 ..
959484 drwxr-xr-x 2 root root 4096 docs
959486 -rw-r--r-- 1 root root 6 notes.txt
959487 lrwxrwxrwx 1 root root 9 readme -> ...
```

Checkpoint: test yourself

- Q1** After `fd2 = dup(fd1)`, you `lseek fd1`. Does `fd2`'s offset move? → **yes: one struct file, one `f_pos`**
- Q2** Where is a file's name stored: in its inode or elsewhere? → **in the directory entry (name → inode number)**
- Q3** You `rm` a file that a process still has open. When is the space freed? → **when the last fd is closed (links 0 and no opens)**
- Q4** Which line of `vfs_read` makes pipes and ext4 work the same way? → **file->f_op->read / read_iter: a function pointer**
- Q5** Why is the second cat of a file faster? → **page cache (data) and dcache (names) are already in RAM**

Key ideas from today



fd (per process) → struct file (offset, f_op) → inode (the file). dup and fork share the middle.



The inode holds everything but the name; st_mode = type + rwx bits; st_nlink counts names.



Directories map names to inode numbers; lookup walks one name at a time, cached in the dcache.



The VFS calls through tables of function pointers: ext4, pipes, /proc and devices plug in.

Homework 12.1 (due before Lecture 12.2)

1. Two writers

append.c writes 1000 lines from a parent and 1000 from a child into one file, in 3 modes: open2, append, fork. Predict the line count of each; run it; explain with the three tables.

2. Link counts

Predict st_nlink of a new empty directory, after mkdir of 3 subdirectories in it, and after creating 3 files in it. Check with stat -c %h.

3. Sorted myls

Make myls sort the names with qsort (write the compare function). Run it on /dev and /proc/self/fd: explain the c, b and l letters, and the fd you never opened.

4. Space not freed

Make a 500 MiB file, keep it open with `sleep 30 < big &`, rm it, and watch df. When does the space come back, and how can you find such files?

5. Read the source

In fs/read_write.c, which two pointers can vfs_read call? Which one do ext4, pipes, /proc and /dev/null use? (fs/ext4/file.c, fs/pipe.c, fs/proc/base.c)

Next lecture and reading

NEXT: LECTURE 12.2

File systems on disk

How ext2/ext4 lay out inodes, directories and blocks on the disk; the page cache and writeback; fsync; what a crash does, and how the journal saves you. Lab: read a file from a raw disk image by hand.

Main text

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., Ch. 10 (system-level I/O) today.

Kernel side

R. Love, **LKD**, Ch. 13 (VFS) today; Ch. 14 (block I/O) and Ch. 16 (page cache and writeback) for 12.2.

Free book

Arpaci-Dusseau, **OSTEP**: "Files and Directories" today; "File System Implementation" and "Crash Consistency" for 12.2.

Lab files

`lab12_files.zip`: `mys_starter.c` and the demos; `ext2cat_starter.c` and `disk.img` for Lab 12.2.