

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 9

Memory and Files

virtual memory, page faults, malloc, and the file system

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

CLASS

9 / 10

19 slides, taken from the full course:
Lecture 11.1, Lecture 11.2, Lecture 12.1

Five problems one idea solves

Isolation: A process cannot even name another process's memory: its addresses lead only to its own frames.

Same addresses for everyone: Every program can be linked to start at the same place; the stack is always near 0x7ff...

More than RAM: malloc(4 GiB) succeeded on our 8 GiB machine: memory is promised first, given page by page.

Sharing: One copy of libc's code in RAM, mapped into every process (vtop: "only mine 0").

Protection: Each page has rights: code r-x, constants r--, data rw-. Breaking them: SIGSEGV.

DIFFERENT FROM C

Not the RAM address

Two processes can hold the same pointer value and reach different bytes.

Analogy: a city with private maps. Every family gets its own map, and every map has a "House No. 10". The post office (the MMU) looks at whose letter it is, and delivers it to the real plot of land written in that family's map.

SAME AS C

A pointer is a number

In C, `int *p` holds an address, and you can print it with `%p`. That number is a VIRTUAL address: it means something only inside your process.

Book: CS:APP §9.1–9.2; Love, LKD Ch. 15

Where each kind of C variable lives (layout.c)

./layout (real)

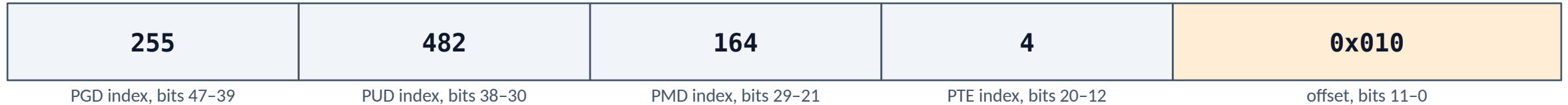
```
main (code)      0x55c131d23140
"hello" rodata   0x55c131d240e1
global_init      0x55c131d26010
global_zero      0x55c131d2602c
malloc(100)     0x55c13284c2a0
malloc(1 MiB)   0x7fd2e42ff010
printf (libc)    0x7fd2e4460100
local (stack)   0x7ffd57af587c
```

Each line is a **VMA** (virtual memory area): a range with one set of rights and one source (a file, or nothing = zeros). The heap grows up with `brk`; big mallocs get their own mapping; the stack grows down from near `0x7fff...`

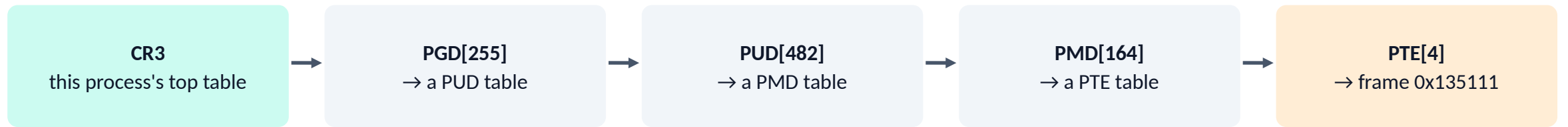
/proc/self/maps of the same run (real, shortened)

```
55c131d22000-55c131d23000 r--p layout # ELF headers
55c131d23000-55c131d24000 r-xp layout # code
55c131d24000-55c131d25000 r--p layout # rodata
55c131d26000-55c131d27000 rw-p layout # data, bss
55c13284c000-55c13286d000 rw-p [heap]
7fd2e42ff000-7fd2e4400000 rw-p # 1 MiB malloc
7fd2e4400000-7fd2e4428000 r--p libc.so.6
7fd2e4428000-7fd2e45b1000 r-xp libc.so.6
7fd2e4604000-7fd2e4606000 rw-p libc.so.6
7fd2e46b5000-7fd2e46b7000 r-xp [vdso]
7fd2e46b7000-7fd2e46b8000 r--p ld-linux-x86-64.so.2
7ffd57ad8000-7ffd57afe000 rw-p [stack]
ffffffffffff600000-...601000 --xp [syscall]
```

Translating an address: four levels of 512 entries



virtual address 0x7ff894804010 (vtop.c, real); the CPU uses its lower 48 bits



```
arch/x86/include/asm, Linux 6.8 (real lines)
```

```
page_types.h      10  #define PAGE_SHIFT      12
pgtable_64_types.h 75  #define PGDIR_SHIFT     39
                  84  #define PUD_SHIFT      30
                  91  #define PMD_SHIFT      21
                  97  #define PTRS_PER_PTE  512
```

C link: like `pgd[255][482][164][4]`, but each inner array exists only if the process uses that part of its address space: a tree, not a full 4-D array.

Why not one flat table? 2^{36} pages $\times$ 8 bytes = 512 GiB per process. With the tree, 100 MiB of touched memory needs about 200 KiB of tables.

Book: CS:APP §9.6.3 (multi-level page tables), §9.7.1

Demand paging: malloc promises, the first touch delivers

● ● ● ./rss (our machine, real)

start	VmRSS	1412 kB	page faults:	89 minor, 0 major
after malloc(100 MiB)	VmRSS	1792 kB	page faults:	92 minor, 0 major
after touching 50 MiB	VmRSS	52988 kB	page faults:	12891 minor, 0 major
after touching 100 MiB	VmRSS	104188 kB	page faults:	25691 minor, 0 major
touching it all again	VmRSS	104188 kB	page faults:	25691 minor, 0 major
after free	VmRSS	1788 kB	page faults:	25691 minor, 0 major

malloc(100 MiB): VmRSS (RAM really used) hardly moves: only the VMA was created.

Touch 50 MiB: 12 800 more faults: one per 4 KiB page, the first time it is touched.

Touch again: No faults: the pages are present now.

free: The mapping is removed (munmap): RAM goes back at once.

● ● ● QEMU guest, Ubuntu 6.8, 512 MiB (real)

after malloc(100 MiB)	VmRSS	860 kB	page faults:	55 minor
after touching 100 MiB	VmRSS	103256 kB	page faults:	25654 minor
after free	VmRSS	856 kB	page faults:	25654 minor

25 600 faults = 100 MiB / 4 KiB. Each one is the path in the next video: about a microsecond each, invisible to the program.

Book: CS:APP §9.3.4 (demand paging)

Watch one page fault, from the MMU to the kernel and back

A page fault: the first write to a new page

SPLITWALKFAULTFIXRETRY

```
user: big[5 * 4096] = 1;
movb $1, (%rax)    rax = 0x7ff894804010
```

255

bits 47-39

482

bits 38-30

164

bits 29-21

4

bits 20-12

0x010

bits 11-0

4 indices of 9 bits + a 12-bit offset (inside the 4 KiB page)

PGD[255] → the next table

PUD[482] → the next table

PMD[164] → the next table

PTE[4] NOT PRESENT

CR3 → PGD → PUD → PMD → PTE: four memory reads (the TLB usually saves them)

kernel (Linux 6.8)

exc_page_fault
fault.c 1510: address from CR2

do_user_addr_fault
fault.c 1230

lock_vma_under_rcu
1347: a VMA rw-p covers it

handle_mm_fault
1355 → memory.c 5438

do_anonymous_page
memory.c 4242: zeroed page

RAM (frames)

free frame

handle_mm_fault finds the missing PTE in an anonymous mapping: do_anonymous_page takes a free frame and fills it with zeros, so you never see another program's old data.

Watch for

1. The address is **cut into 5 parts**.
2. The walk ends at a PTE that is **not present**.
3. The kernel checks the **VMA** first.
4. A **zeroed** frame is given.
5. The **same instruction** runs again.

Click the video to play or pause. About 1.2 minutes, no sound.

Copy-on-write, counted (cow.c)

./cow (our machine, real)

```
fork took 1.54 ms
child read 100 MiB:      2 page faults    11.5 ms
child wrote 100 MiB: 25600 page faults  90.6 ms
child wrote it again:    0 page faults     0.4 ms
parent still sees p[0] = 1
```

QEMU guest (emulated CPU: slower; real)

```
fork took 6.63 ms
child read 100 MiB:      2 page faults     8.6 ms
child wrote 100 MiB: 25600 page faults 394.1 ms
```

1

fork copies the page tables and marks every page **read-only** in both: fast.

2

Reading needs nothing: the shared frame is there.

3

Writing faults (read-only PTE): `do_wp_page` copies 4 KiB into a new frame, makes it writable.

4

Writing again: the page is the child's own now: no fault.

fork + exec (Lecture 8.1) almost never copies anything: `exec` throws the shared pages away before anyone writes them.

When the fault is a bug: SIGSEGV, with the reason

```
./segv2 (SA_SIGINFO handler; real)
```

```
read *NULL          SIGSEGV at (nil)          si_code 1 = SEGV_MAPERR (nothing mapped there)
read 0x123456789000  SIGSEGV at 0x123456789000    si_code 1 = SEGV_MAPERR (nothing mapped there)
write into "hello" (rodata) SIGSEGV at 0x55e575cb608e si_code 2 = SEGV_ACCERR (mapped, but not allowed)
write into the code of try() SIGSEGV at 0x55e575cb5230 si_code 2 = SEGV_ACCERR (mapped, but not allowed)
```

```
segv2.c (the handler)
```

```
static void on_segv(int sig, siginfo_t *si, void *c)
{
    addr = si->si_addr;    /* where */
    code = si->si_code;     /* why */
    siglongjmp(back, 1);    /* not back to the fault */
}
```

SEGV_MAPERR: no VMA contains the address: NULL, a wild pointer, a freed and unmapped block.

SEGV_ACCERR: a VMA exists, but its rights say no: writing to code or string constants, executing data.

DIFFERENT FROM C

Not every C bug faults

Writing past an array into memory that is still mapped gives no fault at all: silent corruption (Lecture 11.2).

Small requests: the heap (brk). Big ones: their own mmap

```
● ● ● strace -e trace=brk,mmap ./mallocoz (real)
```

```
--- malloc(100)
brk(NULL)                = 0x55ffb33e3000
brk(0x55ffb3404000)      = 0x55ffb3404000
--- malloc(1000)         # no system call
--- malloc(100000)       # no system call
--- malloc(200000)
mmap(NULL, 200704, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    MAP_ANONYMOUS, -1, 0) = 0x7f3c61f82000
--- malloc(1048576)
mmap(NULL, 1052672, ...)  = 0x7f3c61e81000
```

```
● ● ● ./mallocoz (the addresses; real)
```

```
malloc( 100) = 0x55ffb33e32a0   malloc(1000)   = 0x55ffb33e4320
malloc(100000) = 0x55ffb33e4710 # all in [heap]
malloc(200000) = 0x7f3c61f82010 # own mapping + 16
```

First malloc: the heap grows by $0x21000 = 132$ KiB at once. The next 100 000 bytes fit in it: no system call.

≥ 128 KiB: glibc gives the block its own anonymous mapping (DEFAULT_MMAP_THRESHOLD_MIN, malloc.c 946). free returns it with munmap at once.

Threshold moves: after you free an mmap'ed block, glibc raises the threshold to its size (up to 32 MiB): Homework 3.

Why two ways? The heap is one contiguous area: a big free hole in its middle cannot be returned. A separate mapping can be returned whole.

The program break: the end of the heap

```
● ● ● include/linux/mm_types.h, Linux 6.8 (real)
```

```
734 struct mm_struct {  
751     struct maple_tree mm_mt; /* the VMAs */  
765     pgd_t * pgd;             /* page tables */  
853     unsigned long total_vm;  
870     unsigned long start_code, end_code, ...  
871     unsigned long start_brk, brk, start_stack;
```

`brk(addr)`

"let my heap end at addr" (mm/mmap.c 178). The kernel grows or shrinks the [heap] VMA; the pages arrive by page fault.

`sbrk(n)`

the libc helper: move the break by n bytes, return the old end. `sbrk(0)` asks where it is.

Our lab's malloc

calls `sbrk` for every new block: 4 system calls above. glibc asks for 132 KiB at once and cuts it.

```
● ● ● strace -e trace=brk ./mymalloc (Lab 11.2; real)
```

```
brk(NULL)                = 0x55b157303000  
brk(0x55b157303090)      = 0x55b157303090 # a  
brk(0x55b1573034a0)      = 0x55b1573034a0 # b  
brk(0x55b157303500)      = 0x55b157303500 # c  
brk(0x55b1573040e0)      = 0x55b1573040e0 # e
```

Book: CS:APP §9.9.1; Love, LKD Ch. 15 (the memory descriptor)

The same ideas in 100 lines: our own malloc at work

Inside a heap: our own malloc and free

MALLOC FREE **SPLIT** MERGE GROW

```
d = my_malloc(200);
```



header: 32 bytes (size, next, free)

in use

free

the heap from its start (+0x000) to the program break (not to scale: small blocks are drawn wider so they stay readable)

`d = malloc(200)`: first fit finds b's free block (1008 bytes), takes 208 of them and splits the rest into a new free block with its own header: $1008 - 208 - 32 = 768$.

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

Watch for

1. Every block has a **header**.
2. free only **marks** a block.
3. **First fit + split**.
4. Free neighbours **merge**.
5. Only when nothing fits, the **break** moves.

"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

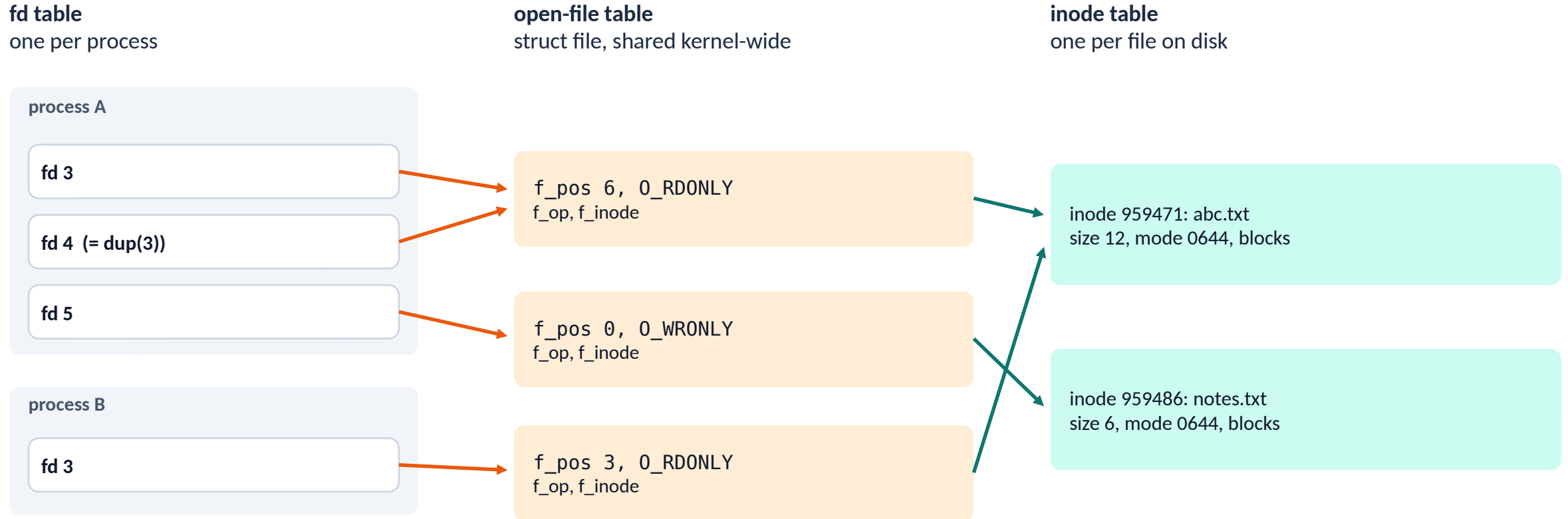
In C you used `fopen` for files on disk. In Linux the same `int fd` can name a pipe, a socket, a terminal, or kernel data.

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

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



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

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

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

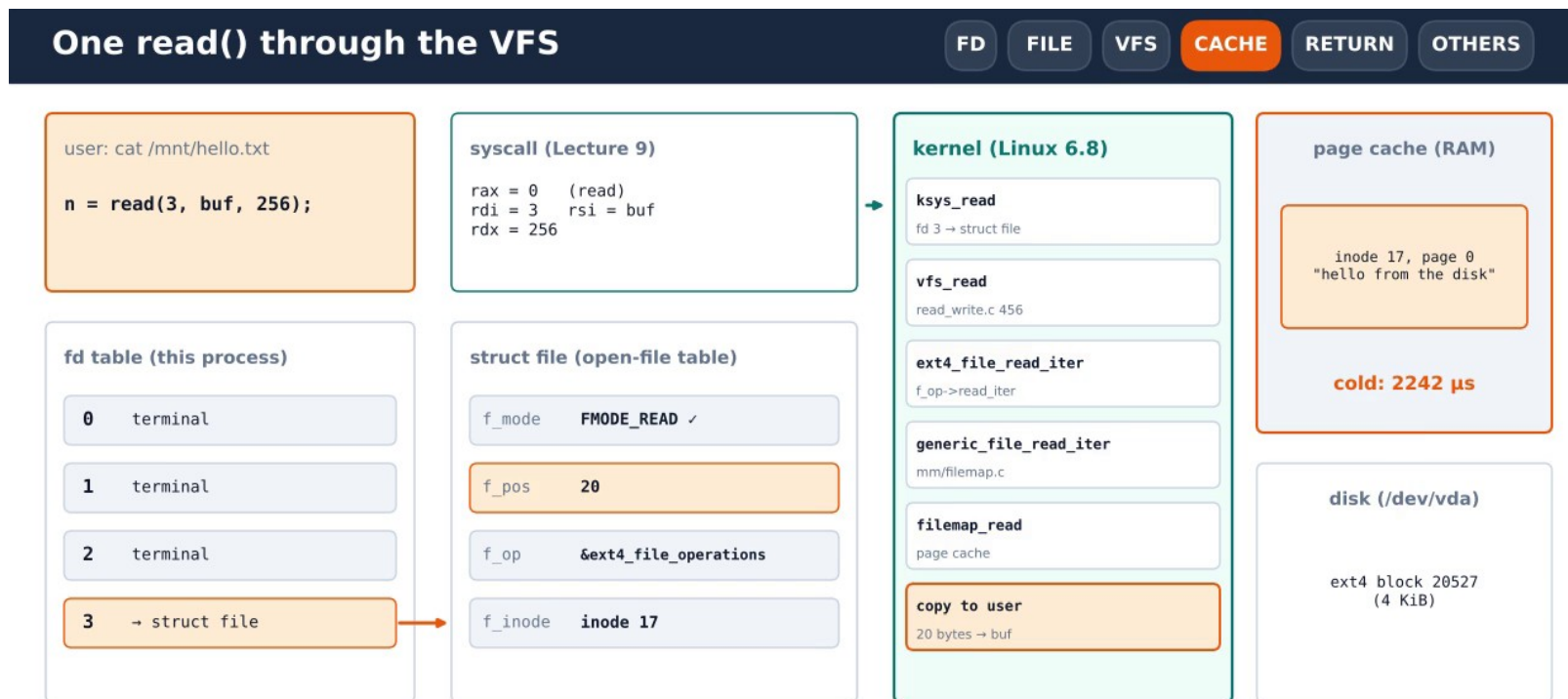
fttrace in the QEMU guest after dropping the caches (real; fs/namei.c)

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)

Watch one read() go through the VFS



The block arrives in a page of the page cache. filemap_read copies 20 bytes from that page into the user's buf and moves f_pos from 0 to 20. This cold read took 2242 µs in our (emulated) guest.

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**.

Key ideas from Class 9

- 1 Every process sees its own virtual addresses; page tables map them to physical frames of 4 KiB.
- 2 A page fault is how the kernel gives memory on first touch, shares pages after fork (copy-on-write) or reports a bug (SIGSEGV).
- 3 malloc takes small blocks from the heap (moved with brk) and big ones from mmap.
- 4 An fd points to an open file, which points to an inode: the inode holds everything about a file except its name.
- 5 A directory is a table of names and inode numbers; the VFS lets one read() work on every file system.

Homework and practice

Homework 9 (bring it to the next class)

1. Split the address 0x7ffd3c2a1b40 into the four page-table indices and the offset (9, 9, 9, 9 and 12 bits).
2. Why does `malloc(1000000)` not use any RAM until you write to it?
3. What does `ls -li` show? What changes, and what does not, when you rename a file?

Try it in the browser

Address → page tables, and a malloc heap

[labs/lab11.html](#)

On the course website: open the lab page and choose "Try it here".

Next class: Interrupts, Modules and Drivers