

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 10

Interrupts, Modules and Drivers

how hardware gets the kernel's attention, and your own code inside the kernel

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

CLASS

10 / 10

21 slides, taken from the full course:

Lecture 13, Lecture 14.1, Lecture 15.1, Lecture 15.3

Every event that stops the CPU is one of four kinds

class	cause	sync?	returns to	examples (x86-64 Linux)
interrupt	a device, a timer, another CPU	async	the next instruction	disk done, key pressed, timer tick
trap	an instruction, on purpose	sync	the next instruction	syscall, int3 (breakpoint)
fault	an instruction that might be fixed	sync	the SAME instruction, again	page fault (Lecture 11.1), #GP, #UD
abort	a fatal hardware error	sync	never	machine check, double fault

A student raises a hand: to ask the teacher something: a **trap** (system call). Planned, at a known moment.

A student makes a mistake: the teacher must look at it: a **fault**. Fix it and try the same step again, or fail the student (SIGSEGV).

The phone rings: in the middle of a sentence: an **interrupt**. Finish the sentence, answer, then continue the lesson.

Book: CS:APP §8.1 (exceptions, four classes); Love, LKD Ch. 7

The IDT: an array of 256 handler addresses

arch/x86/include/asm/trapnr.h (real)

```
7  #define X86_TRAP_DE 0 /* Divide-by-zero */
10 #define X86_TRAP_BP 3 /* Breakpoint */
13 #define X86_TRAP_UD 6 /* Invalid Opcode */
15 #define X86_TRAP_DF 8 /* Double Fault */
20 #define X86_TRAP_GP 13 /* General Protection */
21 #define X86_TRAP_PF 14 /* Page Fault */
25 #define X86_TRAP_MC 18 /* Machine Check */
```

arch/x86/kernel/idt.c (real lines)

```
65 SYSG(X86_TRAP_BP, asm_exc_int3),
71 INTG(X86_TRAP_PF, asm_exc_page_fault),
84 static const ... struct idt_data def_idts[] = {
85     INTG(X86_TRAP_DE, asm_exc_divide_error),
88     INTG(X86_TRAP_UD, asm_exc_invalid_op),
94     INTG(X86_TRAP_GP, asm_exc_general_protection),
108    ISTG(X86_TRAP_MC, asm_exc_machine_check, ...),
```

0 – 31 exceptions, fixed by Intel

32 – 235 device interrupts (FIRST_EXTERNAL_VECTOR 0x20)

0x80 old 32-bit int \$0x80 system call (Lecture 9)

0xec – 0xff local timer 0xec, IPIs (reschedule 0xfd), spurious 0xff

C link: think of `void (*idt[256])(void);`. The CPU does `idt[vector]()` by itself. The kernel fills the table at boot; the IDTR register holds its address (lidt).

Book: Bovet & Cesati Ch. 4 (interrupts and exceptions); Love, LKD Ch. 7

What the CPU does by itself, before any kernel code runs

- 1 Finish the current instruction (interrupt) or stop it (fault)
- 2 Look up IDT[vector]; check the gate's privilege level
- 3 From user mode: switch to this thread's **kernel stack** (address from the TSS)
- 4 Push **SS, RSP, RFLAGS, CS, RIP** (and an error code for some exceptions)
- 5 Interrupt gate: clear **IF** (no more interrupts on this CPU); jump to the handler
- 6 Handler done: **iretq** pops the five values and the program continues

```
arch/x86/include/asm/ptrace.h (real)

59 struct pt_regs {
    ... r15 ... ax: saved by the kernel's entry code
84     unsigned long orig_ax; /* err code, IRQ, nr */
85     /* Return frame for iretq */
86     unsigned long ip;
87     unsigned long cs;
88     unsigned long flags;
89     unsigned long sp;
90     unsigned long ss;
```

SAME AS C

A function call, with more saved

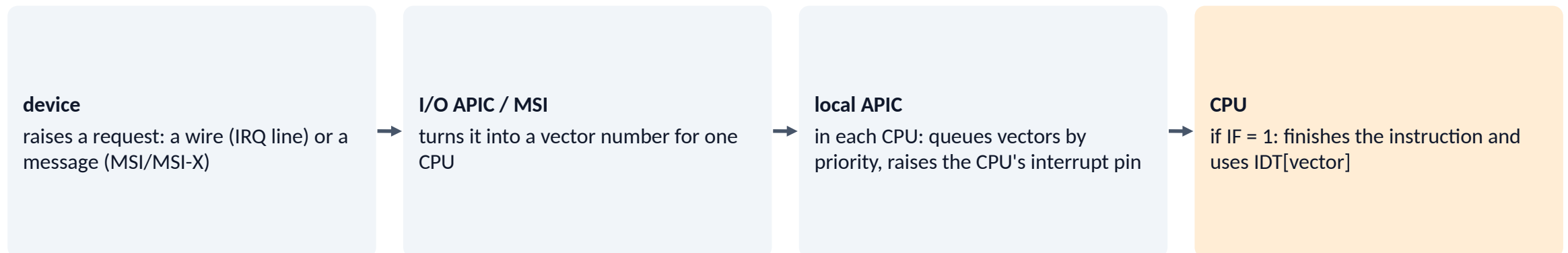
call pushes only the return address; an exception pushes 5 values and switches stacks. **iretq** is the matching **ret**. And as with **setjmp/longjmp**, the saved registers can be changed before returning (Lecture 10.2's signal frames).

Book: Bovet & Cesati §4.2; Intel SDM Vol. 3, Ch. 6

Polling or interrupts: who waits for whom?

	polling	interrupts
how	the CPU asks again and again: done?	the device tells the CPU when done
CPU while waiting	busy, asking	free: runs other programs, or sleeps (hlt)
delay after "done"	up to one polling interval	a few microseconds
good for	very fast devices (NVMe, 100 Gbit)	keyboard, disk, timer, USB, ...

Analogy: waiting for a parcel. Polling: you walk to the gate every five minutes to look. Interrupts: the courier rings your phone. You can study in between.



Book: Love, LKD Ch. 7; Bovet & Cesati §4.2

/proc/interrupts: every interrupt, counted per CPU

```
cat /proc/interrupts (QEMU guest, 1 CPU; real, shortened)
```

```

CPU0
 0: 579 IO-APIC 2-edge timer
 1: 9 IO-APIC 1-edge i8042
 4: 18 IO-APIC 4-edge ttyS0
 8: 1 IO-APIC 8-edge rtc0
14: 0 IO-APIC 14-edge ata_piix
25: 10 PCI-MSIX-0000:00:04.0 1-edge virtio0-req.0
NMI: 0 Non-maskable interrupts
LOC: 2328 Local timer interrupts
RES: 0 Rescheduling interrupts
CAL: 0 Function call interrupts
TLB: 0 TLB shutdowns
```

number: the IRQ number (not the vector); one column of counts per CPU

IO-APIC / PCI-MSIX who delivers it; edge or level ("fastest") triggering

the name given by the driver in request_irq: virtio0-req.0 is our disk, ttyS0 the serial console

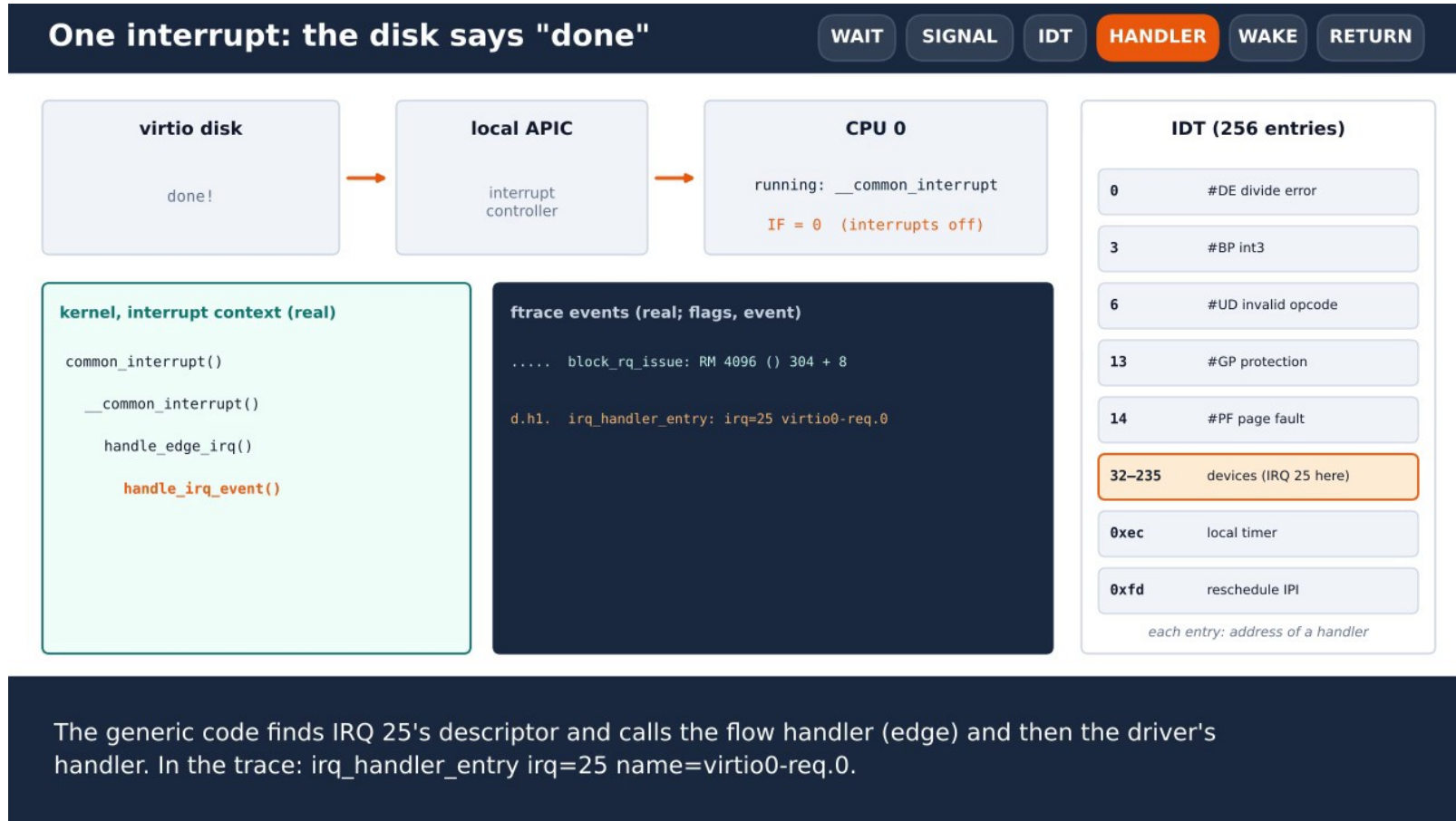
LOC, RES, CAL, TLB CPU-local: timer ticks, and messages between CPUs (IPIs)

Our 2-CPU test machine shows the same kinds of lines: virtio1-req.0 had **1973** interrupts on CPU 1.

On a lab PC: ahci or nvme, xhci_hcd (USB), i915 ...

Book: Love, LKD Ch. 7 (/proc/interrupts)

Watch one interrupt, from the disk to the woken process



Click the video to play or pause. About 1.4 minutes, no sound.

Watch for

1. The CPU **sleeps** in hlt; no polling.
2. APIC → **vector** → IDT entry.
3. **IF = 0** while entering.
4. The driver's handler only **completes and wakes**.
5. **iret** or a switch to readone.

Interrupt context: be fast, never sleep

in a hard interrupt handler	why
no sleeping: no mutex_lock, no msleep	"current" is just whoever was interrupted
no copy_to_user / copy_from_user	they can page-fault and sleep; and whose memory would it be?
kmalloc only with GFP_ATOMIC	GFP_KERNEL may wait for memory to be freed
keep it short (microseconds)	IRQs are off on this CPU: other devices must wait

Analogy: a restaurant. The waiter takes your order at once and writes it on a slip (top half: fast, must not be missed). The kitchen cooks it a few minutes later (bottom half: the long work). If the waiter cooked every dish himself, the next customers would wait at the door.

1 Top half (the handler): acknowledge the device, take the data or the status, schedule the rest. Interrupts off.

2 Bottom half: everything else, later, with interrupts on: softirqs, tasklets, workqueues, threaded IRQs.

Same as Lecture 10.2: a signal handler only sets a flag; an interrupt handler does only the minimum. Same reason: it interrupts code at an unknown point.

Book: Love, LKD Ch. 7 (interrupt context), Ch. 8 (bottom halves)

A module is a piece of the kernel you can plug in while it runs

Small kernel: Ubuntu's kernel image holds only what every PC needs; drivers for the rest are modules on disk.

Only what you have: When the kernel finds a device, udev loads the matching module automatically (Lecture 14.2).

Fast development: Change, rebuild, rmmod, insmod: seconds, not a reboot.

Same power as the kernel: A module runs in kernel mode with full rights. No protection at all: one bad pointer stops the machine.

Our rule: build on the lab PC, **load only in the QEMU guest** (Lab 7's VM). A crash there costs one reboot of the VM.

Analogy: a phone and its apps? Closer: a car and a new engine part fitted **while driving**. It works at full power, and if it is faulty the whole car stops.

SAME AS C

Like a shared library

Lecture 5.2: code loaded at run time, symbols resolved then. A module is the same idea for the kernel.

DIFFERENT FROM C

No process around it

A .so lives in one process's memory. A module lives in the kernel, shared by everything.

Book: Love, LKD Ch. 17 (modules); LDD3 Ch. 2

hello.c and its Makefile

```
hello.c

#include <linux/init.h>
#include <linux/module.h>

static int __init hello_init(void)
{
    pr_info("hello: loaded into the kernel, HZ = %d\n", HZ);
    return 0;    /* < 0: insmod fails with that errno */
}

static void __exit hello_exit(void)
{
    pr_info("hello: removed\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

```
Makefile (Kbuild)

obj-m += hello.o
KDIR ?= /lib/modules/$(shell uname -r)/build
all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
```

No main(): the kernel calls `hello_init` when loading, `hello_exit` when removing

No printf: `pr_info` writes to the kernel log, read with `dmesg`

__init: the code is freed after init: it runs once

Book: LDD3 Ch. 2; Love, LKD Ch. 17

Build it, load it, look at it, remove it

● ● ● make (real, shortened)

```
make -C /usr/src/linux-headers-6.8.0-146-generic M=... modules
CC [M]  hello.o
MODPOST Module.symvers
CC [M]  hello.mod.o
LD [M]  hello.ko
warning: the compiler differs ... (only the name differs)
```

● ● ● lsmod; cat /proc/modules (real)

Module	Size	Used by	Tainted: G
hello	12288	0	
hello 12288 0 - Live 0xffffffffc030e000 (OE)			

● ● ● QEMU guest, as root (real)

```
$ insmod hello.ko
hello: loading out-of-tree module taints kernel.
hello: module verification failed: signature ... missing
hello: loaded into the kernel, HZ = 1000
$ rmmod hello; dmesg | tail -1
[ 8.467078] hello: removed
```

● ● ● ls /sys/module/hello; cat ... (real)

coresize	holders	initsize	initstate	notes
refcnt	sections	srcversion	taint	uevent
initstate: live				
refcnt: 0				
taint: OE				
coresize: 12288				

Watch insmod hello.ko, step by step

insmod hello.ko, step by step

READCHECKPLACESYMBOLS**RELOCATE**INIT

hello.ko (ELF, type REL)

.init.text	hello_init
.exit.text	hello_exit
.rodata	"hello: loaded ..."
.modinfo	vermagic=6.8.0-146-generic ...
__versions	_printk 0x122c3a7e ...
.rela.init.text	call _printk at +0x26

\$ insmod hello.ko

```
fd = open("hello.ko")
finit_module(fd, "", 0)
```

kernel: load_module()

1. finit_module(fd): syscall 313
2. check ELF, vermagic, signature
3. allocate kernel memory, copy sections
4. look up undefined symbols
- 5. apply relocations**
6. call init: do_one_initcall

kernel memory (real addresses)

```
ffffffff929b84b0 T _printk (the kernel)

fffffffc0314010 hello_init (.init.text)
+0x25: e8 86 44 6a d2
call _printk

fffffffc030e000 hello_exit (.text)
fffffffc0310040 __this_module (.data)

hello: module verification failed:
signature ... missing - tainting kernel
```

The relocation R_X86_64_PLT32 says: at +0x26, write $S + A - P = \text{ffffffff929b84b0} - 4 - \text{fffffffc0314026}$. The four zero bytes become 86 44 6a d2: the call now reaches `_printk`. Exactly Lecture 5.1's rule.

Watch for

1. `finit_module`: one syscall.
2. `vermagic` and the signature.
3. Sections copied to `0xffffffffc0...`
4. `_printk` found, then the call patched.
5. `init` runs: Live.

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

Three kinds of device, two of them in /dev

kind	talks in	seek?	examples	in /dev?
character	a stream of bytes	maybe	serial port, keyboard, /dev/null, a sensor	c ...
block	fixed blocks (512 B sectors)	always	disks, SSDs, USB sticks, our RAM disk (15.2)	b ...
network	packets	no	eth0, wlan0	no: sockets instead

```
ls -l /dev/... (QEMU guest, real)
crw----- 1 0 0 1, 1 /dev/mem
crw-rw-rw- 1 0 0 1, 3 /dev/null
crw-rw-rw- 1 0 0 5, 0 /dev/tty
crw----- 1 0 0 4, 64 /dev/ttyS0
crw-rw-rw- 1 0 0 1, 9 /dev/urandom
crw-rw-rw- 1 0 0 1, 5 /dev/zero
```

SAME AS C

open, read, write, close

The same four calls you use on a text file (Lecture 12.1) work on /dev/null and /dev/ttyS0.

Where the size column usually is, ls shows **major, minor**: these files hold no data at all, only two numbers.

Book: LDD3 Ch. 1 (classes of devices) and Ch. 3

dev_t: major and minor packed into one number

```
include/linux/kdev_t.h (real)

7 #define MINORBITS 20
8 #define MINORMASK ((1U << MINORBITS) - 1)
10 #define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
11 #define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
12 #define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

241

major (12)

0

minor (20)

$\text{MKDEV}(241, 0) = 241 \ll 20 = \text{0x0F100000}$

C link: shifts and masks, exactly like the permission bits of Lecture 12 and the ioctl numbers later today.

```
cat /proc/devices (real, start)
```

Character devices:

```
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
10 misc
13 input
241 sysbuf # after insmod sysbuf.ko
```

Fixed or free? Old drivers own a fixed major (1, 4, 5). New drivers ask the kernel for a free one: we got **241**. Never hard-code it: read it from /proc/devices or sysfs.

Book: LDD3 Ch. 3 (major and minor numbers)

Four steps make /dev/sysbuf appear (sysbuf_init)

```
sysbuf.c: init (core lines)

static dev_t devno; static struct cdev cdev; static struct class *cls;

err = alloc_chrdev_region(&devno, 0, 1, "sysbuf"); /* free major */
if (err) return err;
cdev_init(&cdev, &sysbuf_fops);
err = cdev_add(&cdev, devno, 1); /* open() can reach us */
if (err) goto undo_region;
cls = class_create("sysbuf"); /* /sys/class/sysbuf */
if (IS_ERR(cls)) { err = PTR_ERR(cls); goto undo_cdev; }
device_create(cls, NULL, devno, NULL, "sysbuf"); /* uevent */
return 0;
undo_cdev: cdev_del(&cdev);
undo_region: unregister_chrdev_region(devno, 1);
return err;
```

1

a number: alloc_chrdev_region → 241:0 (char_dev.c 236)

2

a cdev: our file_operations, added to the kernel's table for 241:0 (cdev_add, line 479)

3

a class: a directory in /sys/class (Lecture 14.2)

4

a device: sends the uevent; devtmpfs makes /dev/sysbuf

exit does the same four, **in reverse**.

Book: LDD3 Ch. 3 (char device registration)

struct file_operations: the driver's answer to every system call

include/linux/fs.h (real, part)

```
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 *, size_t, loff_t *);
1987     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1994     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    ... open, release, mmap, poll, fsync ... (about 30 in all)
```

sysbuf.c (real)

```
static const struct file_operations sysbuf_fops = {
    .owner      = THIS_MODULE,
    .open       = sysbuf_open,
    .read       = sysbuf_read,
    .write      = sysbuf_write,
    .llseek     = default_llseek,
    .unlocked_ioctl = sysbuf_ioctl,
};
```

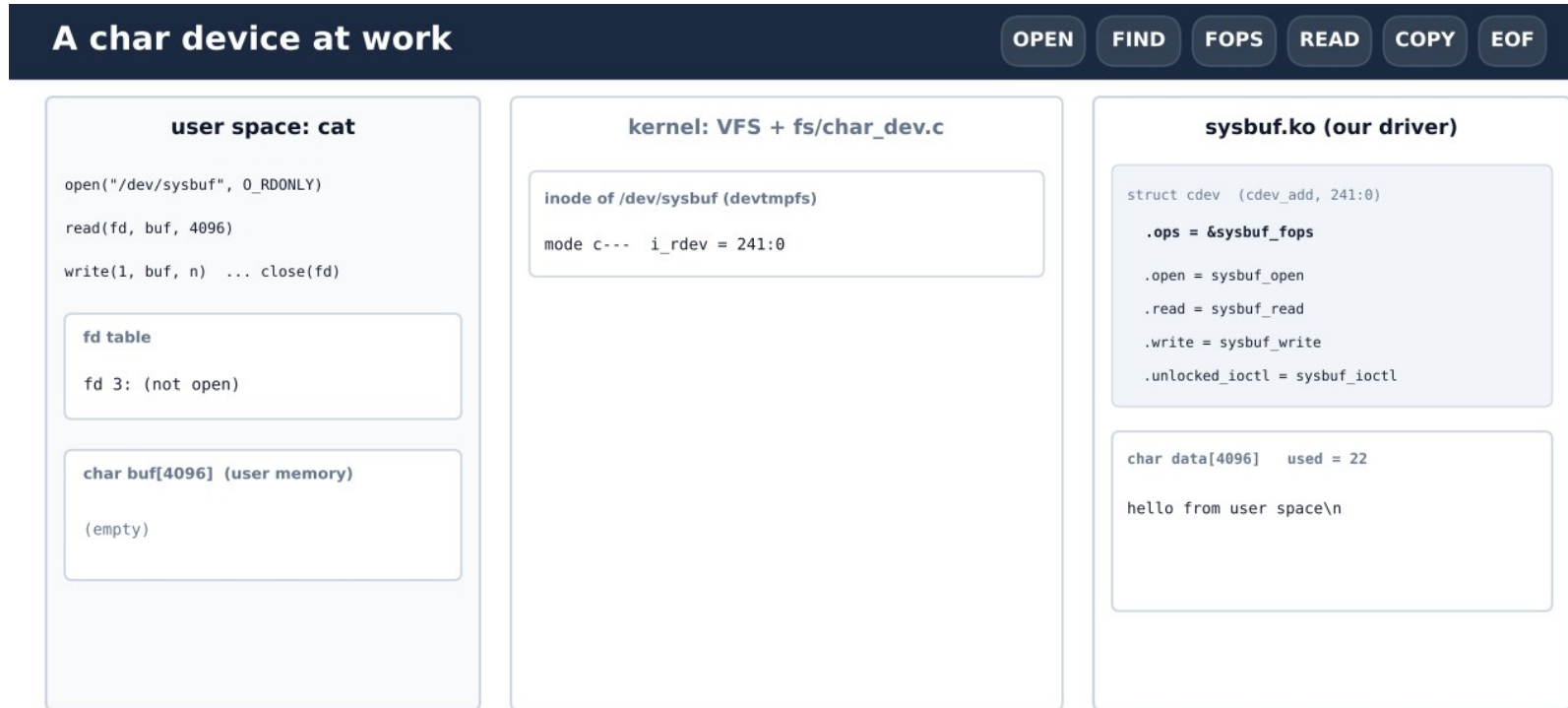
user calls	kernel calls
open("/dev/sysbuf", ...)	sysbuf_open
read(fd, buf, n)	sysbuf_read
write(fd, buf, n)	sysbuf_write
lseek(fd, 14, SEEK_SET)	default_llseek
ioctl(fd, cmd, arg)	sysbuf_ioctl
close(fd)	.release (none: nothing to do)

SAME AS C

A table of function pointers

Like qsort's comparator, or a struct of callbacks in your own C: the kernel calls `f->f_op->read(...)` without knowing who wrote it. Missing entries are NULL; the kernel then uses a default or returns an error.

Watch cat /dev/sysbuf, step by step



After echo hello from user space > /dev/sysbuf, we run cat /dev/sysbuf. The file in /dev is only a name with a type (c = character device) and two numbers, major 241 and minor 0. All the behaviour comes from the driver that registered 241:0.

Watch for

1. The inode only holds **241:0**.
2. **kobj_lookup** finds the cdev.
3. **replace_fops**: f_op becomes ours.
4. **copy_to_user** moves 22 bytes.
5. read returns **0**: EOF.

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

Why copy_to_user, and never memcpy into buf

	memcpy(buf, data, n)	copy_to_user(buf, data, n)
buf is NULL	oops in the kernel	returns n (not copied) → -EFAULT
buf points into the kernel	user overwrites kernel memory!	rejected: not a user address
buf's page is swapped out	page fault in kernel code	fault handled, page brought in
SMAP on (modern x86)	fault: kernel may not touch user pages	opens the user window just for the copy

The rule: a pointer that came from user space is a **number the user chose**. Check it and copy through it with copy_to_user, copy_from_user, get_user, put_user. They return the number of bytes **not** copied (0 = success).

```
● ● ● hwtest151: bad pointers (real)
```

```
write 3 bytes: 3
read into (nil): -1
               (Bad address)
read into 0xffffffff81000000: -1
               (Bad address)
write from 0xffffffff81000000: -1
               (Bad address)
```

Lecture 9 and 13 link: "Bad address" is EFAULT. The kernel survived, because copy_to_user's faults go to a fix-up table instead of an oops.

Book: LDD3 Ch. 3 (read and write); Love, LKD Ch. 5 (system calls)

One key press, through everything we studied

- 1 the host controller polls, the keyboard answers: **an interrupt** on IRQ 11 (Lecture 13)
- 2 the completion runs **driver code**: a module the kernel linked at run time (14.1, like 5.1's loader)
- 3 the driver found its device on a **bus** by matching an `id_table` (14.2, 15.3)
- 4 the key reaches a program through a **char device** `read()` (15.1, 12.1's file operations)
- 5 that program is a **process** whose system calls (9) run kernel code; its ELF (6) was assembled (3) and linked (5)
- 6 and every instruction on that path is **fetch, decode, execute** (Lecture 1)

System programming is the software that makes the machine usable by other software: translators, loaders, the kernel, drivers.

The habit to keep: never guess what the system does: **measure it**. `strace`, `ftrace`, `/proc`, `/sys`, `objdump` and `readelf` answered every question in this course with real output.

Key ideas from Class 10

- 1 Exceptions come from the CPU (divide by zero, page fault); interrupts come from devices; both go through the IDT.
- 2 An interrupt handler must be short and never sleep; the rest of the work is done later.
- 3 A kernel module is code you load into a running kernel with `insmod` and remove with `rmmod`.
- 4 A character driver fills `struct file_operations`; `open`, `read` and `write` on `/dev/name` call those functions.
- 5 `copy_to_user` and `copy_from_user` move data across the user-kernel border safely.

Homework and practice

Homework 10 (bring it to the next class)

1. Polling or interrupts: which is better for a keyboard, and why?
2. What do `module_init` and `module_exit` do? What goes wrong if exit forgets to undo init?
3. What happens, step by step, when a program runs `cat /dev/sysbuf` ?

Try it in the browser

Page-fault error codes and interrupt counts

[labs/lab13.html](#)

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

The end of the short course. The full course on the same website has every detail and all 15 labs.