

# Lecture 15.1

Character Device Drivers  
your own file in /dev

Md Mahbub E Noor

Assistant Professor, Department of CSE  
University of Barishal

```
● ● ● QEMU guest (real)
$ insmod sysbuf.ko
$ echo hello from user space >
/dev/sysbuf
$ cat /dev/sysbuf
hello from user space
# the bytes lived in a C array inside
the kernel
```

# What you will learn today



Explain how a device file (c, major, minor) leads the kernel to one driver.



Write a char driver: register a number, a cdev and file\_operations; make /dev/sysbuf.



Implement read and write with copy\_to\_user / copy\_from\_user, f\_pos and EOF.



Design ioctl commands with \_IO/\_IOR and use them from a C program.

## Today's plan (60 min)

0-5

Recap of 14.2

5-12

Device files

12-24

Writing the driver

24-38

open, read, write

38-48

ioctl

48-60

Lab, checkpoint

*Lab 15.1: sysbuf.c, a 4 KiB device.*

# Quick recap of Lecture 14.2

Q1

Is `ls -l /dev/vda` shows "b ... 253, 0". What do b, 253 and 0 mean?

Block device; major 253 picks the driver (virtblk), minor 0 picks the device (the first disk).

Q2

Who calls a driver's probe function, and when?

The bus, when its match function pairs the driver with a device (platform: by name).

Q3

Who makes the file in `/dev` when a device appears?

devtmpfs (and udev), from the kernel's uevent: MAJOR, MINOR, DEVNAME.

## PART 1

# Devices are files

A name, a letter and two numbers: the kernel does the rest.

# 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)

# The name does not matter; the numbers do (real experiment)

```
● ● ● QEMU guest, as root (real)
$ insmod sysbuf.ko; echo "made by mknod" > /dev/sysbuf
$ mknod /tmp/other c 241 0
crw-r--r-- 1 0 0 241, 0 /tmp/other
$ cat /tmp/other
made by mknod
$ mknod /tmp/minor1 c 241 1; cat /tmp/minor1
cat: can't open '/tmp/minor1': No such device or address
$ rmmod sysbuf; cat /tmp/other
cat: can't open '/tmp/other': No such device or address
$ ls -l /tmp/other
crw-r--r-- 1 0 0 241, 0 /tmp/other # still there
```

1 any name, in any directory: same major and minor, same driver, same data

2 minor 1: the driver registered only minor 0 → ENXIO

3 driver gone: the inode stays, but nobody owns 241 → ENXIO

**Lecture 12.1 link:** the inode of a device file stores the type and `i_rdev = 241:0` instead of block pointers. `open()` looks the number up; the path is forgotten.

## PART 2

# Writing a char driver

sysbuf: a 4 KiB buffer in the kernel that behaves like a small file.

# 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.

# After insmod sysbuf.ko: the number, the node, the sysfs entry (real)

QEMU guest (real)

```
$ insmod sysbuf.ko; dmesg | tail -1
sysbuf: ready as /dev/sysbuf, major 241 minor 0
$ ls -l /dev/sysbuf
crw----- 1 0 0 241, 0 Oct 15 04:52 /dev/sysbuf
$ grep sysbuf /proc/devices
241 sysbuf
```

the sysfs side (real)

```
$ cat /sys/class/sysbuf/sysbuf/dev
241:0
$ cat /sys/class/sysbuf/sysbuf/uevent
MAJOR=241
MINOR=0
DEVNAME=sysbuf
$ readlink -f /sys/class/sysbuf/sysbuf
/sys/devices/virtual/sysbuf/sysbuf
```

1 device\_create → uevent with **MAJOR=241 MINOR=0 DEVNAME=sysbuf** (the same three lines as the uevent file)

2 devtmpfs makes /dev/sysbuf, mode **crw-----** (root only); on a desktop, udev may then change owner and mode by its rules

## PART 3

# open, read, write

From a file descriptor to our functions, and bytes across the user/kernel border.

# How open("/dev/sysbuf") finds our driver

fs/char\_dev.c (real lines, shortened)

```
373 static int chrdev_open(struct inode *inode, struct file *filp)
381     p = inode->i_cdev;          /* cached from an earlier open? */
382     if (!p) {
386         kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
387         if (!kobj)
388             return -ENXIO;
395         inode->i_cdev = p = new;
        } else if (!cdev_get(p)) /* try_module_get */
        fops = fops_get(p->ops);
412     replace_fops(filp, fops);    /* f_op = &sysbuf_fops */
413     if (filp->f_op->open) {
414         ret = filp->f_op->open(inode, filp);
452 const struct file_operations def_chr_fops = {
453     .open = chrdev_open, ...
```

ftrace: cat /dev/sysbuf (real)

```
chrdev_open() {
    _raw_spin_lock();
    try_module_get();
    _raw_spin_unlock();
    try_module_get();
    module_put();
    sysbuf_open [sysbuf]();
}
```

**The trick:** every char device file starts with the same generic f\_op (def\_chr\_fops). Its open looks up the number, then **swaps in the driver's own table**. From then on read() goes straight to sysbuf\_read.

Book: LDD3 Ch. 3 (the file structure, open)

# 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.

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

## 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.

# sysbuf\_read: give bytes from the current position, 0 at the end

```
sysbuf.c (real)

static ssize_t sysbuf_read(struct file *f, char __user *buf,
                          size_t count, loff_t *pos)
{
    ssize_t n;
    mutex_lock(&lock);
    if (*pos >= used) { mutex_unlock(&lock); return 0; } /* EOF */
    n = min(count, used - (size_t)*pos);
    if (copy_to_user(buf, data + *pos, n)) {
        mutex_unlock(&lock); return -EFAULT;
    }
    *pos += n;
    mutex_unlock(&lock);
    return n;
}
```

**SAME AS C**

**The read() contract of Lecture 12.1**

A regular file obeys exactly these rules. A driver that follows them works with cat, dd, C stdio.

```
ftrace: vfs_read (real)
```

```
vfs_read() {
    rw_verify_area() {
        security_file_permission();
    }
    sysbuf_read [sysbuf]() {
        mutex_lock();
        __check_object_size();
        mutex_unlock();
    }
}
```

1 return n > 0: n bytes read; may be fewer than asked

2 return 0: end of file; cat stops

3 return -errno: read() gives -1 and sets errno

# 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)

# sysbuf\_write, and what a full device says

sysbuf.c (real, core lines)

```
if (*pos >= SYSBUF_SIZE) {
    mutex_unlock(&lock);
    return -ENOSPC;      /* "No space left on device" */
}
n = min(count, SYSBUF_SIZE - (size_t)*pos);
if (copy_from_user(data + *pos, buf, n)) { ... return -EFAULT; }
*pos += n;
if (*pos > used) used = *pos;
return n;               /* may be less than count */
```

QEMU guest (real)

```
$ dd if=/dev/zero of=/dev/sysbuf bs=1024 count=5
dd: error writing '/dev/sysbuf':
    No space left on device
5+0 records in
4+0 records out
$ cat /dev/sysbuf | wc -c
4096
```

1 writes 1-4: 1024 bytes each, f\_pos 1024 → 4096. Write 5: f\_pos is 4096 = SYSBUF\_SIZE → -ENOSPC; dd reports it and stops

2 a write that only partly fits returns the smaller n: the caller must loop (Lecture 12.1's "short write"); stdio's fwrite does that for you

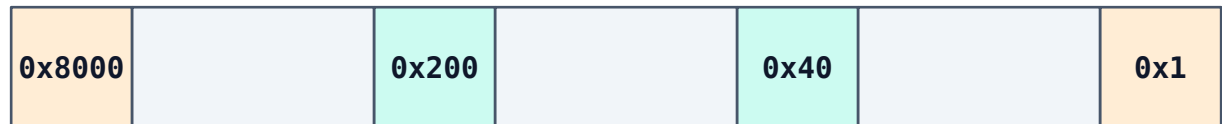
# sysbuf\_open sees the flags: 0x8241 and 0x8000 (real)

```
dmesg (real)

$ echo "hello from user space" > /dev/sysbuf
sysbuf: open by init (pid 1), minor 0, flags 0x8241
$ cat /dev/sysbuf
sysbuf: open by cat (pid 88), minor 0, flags 0x8000
```

```
sysbuf_open (real)

if ((f->f_flags & O_ACCMODE) != O_RDONLY
    && (f->f_flags & O_TRUNC)) {
    mutex_lock(&lock); used = 0;
    mutex_unlock(&lock);
}
```



LARGEFILE (1) ... (2) TRUNC (1) ... (2) CREAT (1) ... (2) WRONLY (1)

0x8241 = O\_LARGEFILE | O\_TRUNC | O\_CREAT | O\_WRONLY (the shell's >)

0x8000 = O\_LARGEFILE | O\_RDONLY (0) (cat)

SAME AS C

**fopen("f", "w") truncates**

The shell's > is open(..., O\_WRONLY|O\_CREAT|O\_TRUNC). For a regular file the file system truncates; for a device, only if the DRIVER decides to: we do.

**O\_CREAT on a device?** The node already exists, so the flag does nothing. The driver sees all flags and may honour the ones that mean something for it (O\_TRUNC, O\_NONBLOCK: Homework 15.1).

# An open file keeps the module loaded: `.owner = THIS_MODULE`

```
QEMU guest (real)
$ cat /sys/module/sysbuf/refcnt
0
$ (sleep 3 < /dev/sysbuf &); sleep 1
$ cat /sys/module/sysbuf/refcnt
1
$ rmmod sysbuf
rmmod: can't unload module 'sysbuf': Resource temporarily unavailable
$ sleep 3; rmmod sysbuf; echo $?
0
```

- 1 sleep's stdin is `/dev/sysbuf`: an open struct file
- 2 `chrdev_open` took a reference: `try_module_get`
- 3 `rmmod` refuses: **EAGAIN** while `refcnt > 0`
- 4 sleep exits, file closed, `module_put`: `rmmod` works

**Without `.owner`**, `rmmod` would succeed while `sleep` still holds the file. Its next read would jump to `sysbuf_read`, which is no longer in memory: an oops, like forgetting `remove_proc_entry` in Lab 14.1. The kernel protects you only if you fill in `.owner`.

## PART 4

# ioctl: everything that is not read or write

Commands with numbers, and a header shared by the driver and its programs.

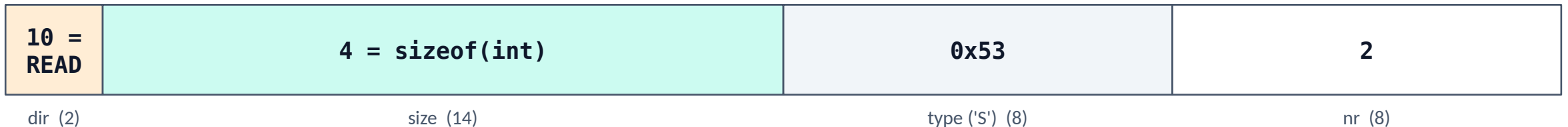
# An ioctl command is a 32-bit number with four fields

● ● ● sysbuf.h (shared by driver and programs)

```
#include <linux/ioctl.h>
#define SYSBUF_MAGIC 'S'
#define SYSBUF_CLEAR _IO(SYSBUF_MAGIC, 1)
#define SYSBUF_GETSIZE _IOR(SYSBUF_MAGIC, 2, int)
```

● ● ● include/uapi/asm-generic/ioctl.h (real)

```
23 #define _IOC_NRBITS      8
24 #define _IOC_TYPEBITS    8
32 #define _IOC_SIZEBITS   14
36 #define _IOC_DIRBITS     2
66 #define _IOC_READ        2U
86 #define _IOR(type,nr,size) _IOC(_IOC_READ,(type),(nr),...)
```



`SYSBUF_GETSIZE = 2 << 30 | 4 << 16 | 0x53 << 8 | 2 = 0x80045302` (sbttest printed exactly this)

**Why encode?** The number says which way data flows and how much, so tools (strace) and the kernel can check it; the magic letter keeps one driver's numbers apart from another's.

Book: LDD3 Ch. 6 (ioctl, choosing the numbers)

# sysbuf\_ioctl, and a C program that uses all of it

sysbuf.c (real)

```
static long sysbuf_ioctl(struct file *f,
                        unsigned int cmd, unsigned long arg)
{
    int size;
    switch (cmd) {
        case SYSBUF_CLEAR:    ... used = 0; ... return 0;
        case SYSBUF_GETSIZE: size = used;
        return put_user(size, (int __user *)arg);
        default: return -ENOTTY;
    }
}
```

./sbtest (real)

```
/* write 22 bytes, GETSIZE, lseek 14, read twice, bad ioctl */
write: 22 bytes, size now 22
read at offset 14: 8 bytes "Barishal"
read again: 0 (end of file)
ioctl 0x1234: Inappropriate ioctl for device
SYSBUF_GETSIZE = 0x80045302 (_IOR('S', 2, int))
```

ftrace: one ioctl (real)

```
__x64_sys_ioctl() {
    __fdget(); security_file_ioctl(); do_vfs_ioctl();
    sysbuf_ioctl [sysbuf]() { mutex_lock(); mutex_unlock(); }
```

ENOTTY = "Inappropriate ioctl for device": the historic name, from the days when ioctl was mostly for terminals. In the same trace, printf's isatty() check went to tty\_ioctl, not to us.

# Two programs, two CPUs, one buffer: why sysbuf has a mutex

| time | CPU 0: writer (echo)                     | CPU 1: reader (cat)                        |
|------|------------------------------------------|--------------------------------------------|
| t1   | *pos 0 < 4096, n = 22                    |                                            |
| t2   | copy_from_user ... (page fault: sleeps!) | *pos 0 < used ?                            |
| t3   |                                          | used is still old: reads half-written data |
| t4   | used = 22                                |                                            |

1 copy\_\*\_user **may sleep** (the user page may need to be read from disk)

2 so the lock must allow sleeping: a **mutex**; a spinlock waiter never sleeps, and a holder must not (Lecture 13)

3 hold it around every access to data and used, and **unlock on every return path**

SAME AS C

## Lecture 10.1's race, again

Two processes, one counter, fixed with a semaphore. Kernel version: `DEFINE_MUTEX(lock); mutex_lock(&lock); ... mutex_unlock(&lock);`

**Forget one unlock** (e.g. on the `-EFAULT` path) and the next reader sleeps forever in `mutex_lock`: the process cannot even be killed with Ctrl-C. Check every return in the lab code.

## LAB 15.1

# sysbuf.c: your own /dev file

Register, read, write, ioctl; test with echo, cat, dd and sbtest.

# Lab 15.1 tasks, step by step

1 `cd lab15_1; make` builds `sysbuf_starter.ko` and the static test program `sbtest`

2 **TODO 1** init: `alloc_chrdev_region`, `cdev_init/cdev_add`, `class_create`, `device_create` (with the goto ladder)

3 **TODO 2** read: EOF, min, `copy_to_user`, `*pos += n`

4 **TODO 3** write: `-ENOSPC`, `copy_from_user`, update used

5 **TODO 4** ioctl: CLEAR, GETSIZE with `put_user`, default `-ENOTTY`

6 `../guest/run_guest.sh` . then `insmod`, `echo`, `cat`, `dd`, `./sbtest`

● ● ● the starter, before **TODO 1** (real)

```
$ insmod sysbuf_starter.ko; echo status $?  
insmod: can't insert 'sysbuf_starter.ko': No such device  
status 19  
$ dmesg | grep sysbuf  
sysbuf: not registered yet: TODO 1
```

● ● ● the finished lab (real)

```
$ ./sbtest  
write: 22 bytes, size now 22  
read at offset 14: 8 bytes "Barishal"  
read again: 0 (end of file)  
ioctl 0x1234: Inappropriate ioctl for device  
SYSBUF_GETSIZE = 0x80045302 (_IOR('S', 2, int))
```

# Checkpoint: test yourself

Q1

ls -l shows "crw----- ... 241, 0 /tmp/x". Which driver runs when you cat it?

→ whoever registered char major 241, minor 0 (sysbuf)

Q2

Your read function returns 0. What does cat do?

→ it sees end of file and stops

Q3

Why may a driver not memcpy into the read buffer?

→ it is a user address: use copy\_to\_user (EFAULT, page faults, SMAP)

Q4

Decode \_IOR('S', 2, int).

→ dir READ, size 4, type 0x53, nr 2 = 0x80045302

Q5

rmmod fails with EAGAIN although nobody uses the driver's API. Why?

→ a process still has the device file open (owner refcount)

# Key ideas from today



A device file is a name for (c, major, minor); the number finds a cdev, the cdev holds file\_operations.



init: number → cdev\_add → class → device\_create (uevent makes /dev); exit undoes it in reverse.



read/write follow the file contract: bytes, f\_pos, 0 = EOF, -errno; copy\_\*\_user for user memory.



ioctl numbers encode direction, size, type and number; unknown commands return -ENOTTY.

# Homework 15.1 (due before Lecture 15.2)

## 1. Blocking read

Make read wait for data instead of returning 0 (`wait_event_interruptible`, `wake_up_interruptible` in write). Show that cat sleeps (state S, wchan). What happens on Ctrl-C, and with `O_NONBLOCK`?

## 2. ioctl numbers

Compute `SYSBUF_CLEAR = _IO('S', 1)` in hex. `printf`'s `isatty()` sent `TCGETS = 0x5401`: decode its four fields. Why was it `_IO`-style, not `_IOR`?

## 3. Open flags

`hwtest151`'s opens printed flags `0x8800` and `0x8202`. Decode both. Which of these flags does `sysbuf_open` act on?

## 4. ENXIO

With `fs/char_dev.c` (lines 373–414) explain both `ENXIO` results of slide 7: minor 1, and the node after `rmmod`.

## 5. copy\_to\_user

Explain each "Bad address" on slide 16, and why the kernel did not oops. What does `copy_to_user` return?

# Next lecture and reading

## NEXT: LECTURE 15.2

### Block device drivers

Disks talk in sectors and requests, not bytes: bios, the request queue (blk-mq), the page cache in front of it. Lab: a RAM disk that you format with ext2 and mount.

#### Driver book

Corbet, Rubini & Kroah-Hartman, **LDD3** (free), Ch. 3 and 6 today; Ch. 16 (block drivers) for 15.2: the ideas, but its API is old.

#### Kernel side

R. Love, **LKD**, Ch. 14 (the block I/O layer).

#### Source

Linux 6.8: fs/char\_dev.c, include/linux/fs.h, cdev.h, kdev\_t.h, uapi/asm-generic/ioctl.h (Linux v6.8 source online: [elixir.bootlin.com/linux/v6.8](https://elixir.bootlin.com/linux/v6.8)).

#### Lab files

`lab15_files.zip`: lab15\_1/ (sysbuf\_starter.c, sysbuf.h, sbtest.c), guest/run\_guest.sh.