

# Lecture 15.2

## Block Device Drivers a disk made of RAM

Md Mahbub E Noor

Assistant Professor, Department of CSE  
University of Barishal

```
● ● ● QEMU guest (real)
$ insmod sbd.ko size_mb=16
$ mke2fs -q /dev/sbd0; mount
/dev/sbd0 /mnt
$ cat /mnt/hello.txt
hello on a RAM disk
# ext2 of Lecture 12.2, on our driver
```

# What you will learn today



Explain how a block device differs from a char device, and why the page cache sits in front.



Follow one write through bio, request, blk-mq and the I/O scheduler to the driver.



Write a blk-mq driver: tag set, gendisk, capacity, queue\_rq with a segment loop.



Watch real I/O with block trace events, `/sys/block/.../stat` and the driver's own log.

## Today's plan (60 min)

- 0-5 Recap of 15.1
- 5-14 Block devices
- 14-26 bio and request
- 26-40 The sbd driver
- 40-48 Watching I/O
- 48-60 Lab, checkpoint

*Lab 15.2: sbd.c, a RAM disk with ext2.*

# Quick recap of Lecture 15.1

**Q1** **sysbuf\_read returns 0. What does cat do, and why?** It stops: 0 means end of file, the same contract as a regular file.

**Q2** **Why copy\_to\_user instead of memcpy into buf?** buf is a user address: it may be NULL or point into the kernel; copy\_to\_user checks it and handles faults (EFAULT).

**Q3** **How does open("/dev/sysbuf") reach sysbuf\_open?** chrdev\_open looks up 241:0 in cdev\_map, replaces f\_op with sysbuf\_fops, then calls f\_op->open.

## PART 1

# Block devices

Fixed-size blocks, random access, and a cache in front.

# Character device vs block device

|             | character (15.1)                 | block (today)                                 |
|-------------|----------------------------------|-----------------------------------------------|
| unit        | bytes, any count                 | sectors of 512 bytes                          |
| access      | usually in order                 | any sector, any time                          |
| driver gets | open, read, write, ioctl calls   | requests: sector, count, memory pages         |
| in between  | nothing: read() calls the driver | page cache, bio, request queue, I/O scheduler |
| used by     | programs directly                | file systems (mount), and programs (dd)       |

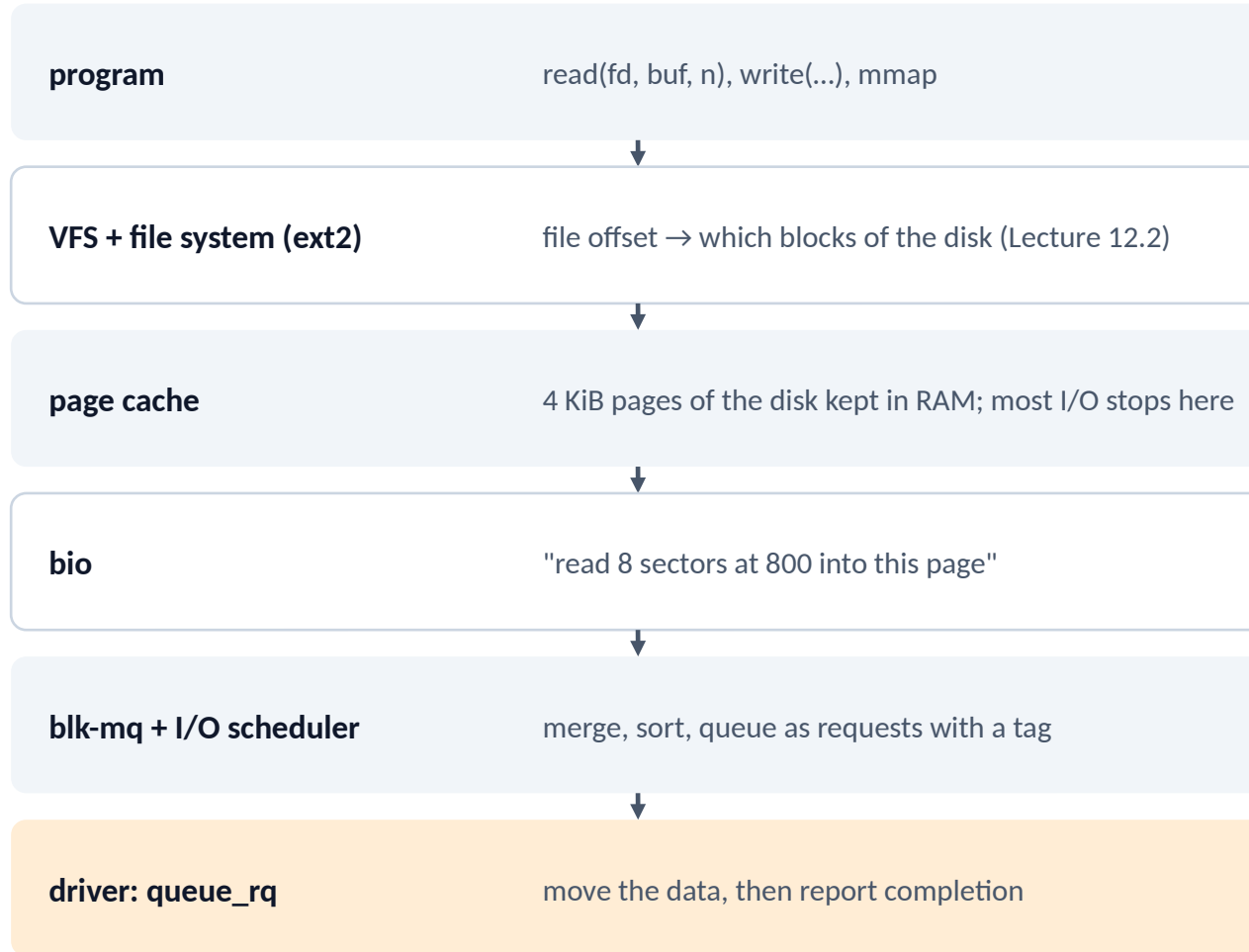
**C link:** a char device is like reading a pipe with read(); a block device is like an array of structs on disk that you index: `disk[800]` is sector 800, always the same 512 bytes, whenever you look.

```
QEMU guest (real)
$ ls -l /dev/sbd0
brw----- 1 0 0 251, 0 /dev/sbd0
$ grep -A9 Block /proc/devices
Block devices:
 7 loop      8 sd
11 sr        251 sbd
252 device-mapper
253 virtblk  254 mdp
```

**b, not c.** The same major/minor idea, but a separate table of majors: block major 251 (sbd) and char major 251 (dimmctl) are different drivers.

Book: Love, LKD Ch. 14 (anatomy of a block device)

# Who stands between a program and a block driver



SAME AS C

## Lecture 12.2: ext2 is an array of blocks

ext2cat read the image file with `pread` at `block * 1024`. Today the same arithmetic happens in the kernel, and the "file" is our driver.

**The driver is at the bottom.** It never sees a file name, an inode or a user pointer: only sectors and pages.

**Why so many layers?** Disks are slow: a seek costs milliseconds. Caching, merging and sorting save far more time than the layers cost.

# The page cache in front of the driver (real experiment)

● ● ● QEMU guest, sbd trace on, /dev/sbd0 held open by fd 3 (real)

```
$ dd if=/dev/zero of=/dev/sbd0 bs=512 count=8 seek=2048
8+0 records out
writes seen while fd 3 is open: 0
$ sync; dmesg | grep 'sbd: WRITE'
sbd: WRITE sector 2048, 4096 bytes
$ dd if=/dev/sbd0 of=/dev/null bs=4096 count=1 skip=512
first read: sbd: READ sector 4096, 4096 bytes
$ dd if=/dev/sbd0 of=/dev/null bs=4096 count=1 skip=512
second read, driver saw: 0
```

1 8 writes of 512 bytes land in **one page** of the cache; the driver sees **nothing**

2 sync writes the dirty page: **one** 4096-byte request

3 the first read fills the cache; the second read **never** reaches the driver

**The trap we fell into first:** without fd 3, the write reached the driver at once and both reads did too. The **last close** of a block device flushes and drops its cache (block/bdev.c 666-669 → blkdev\_flush\_mapping 634). Homework 15.2 task 3.

## PART 2

# bio and request

How the block layer describes I/O, and how it queues it.

# struct bio: "these sectors, these memory pieces"

```
include/linux/blk_types.h, bvec.h (real)

264 struct bio {
267     blk_opf_t      bi_opf;      /* READ, WRITE, flags */
275     struct bvec_iter bi_iter;    /* where we are */
278     bio_end_io_t    *bi_end_io;  /* called when done */
304     unsigned short  bi_vcnt;    /* how many bio_vecs */
314     struct bio_vec  *bi_io_vec; /* the list */
bvec.h
31 struct bio_vec { struct page *bv_page;
33                 unsigned int bv_len, bv_offset; };
78 sector_t bi_sector; /* device address in 512 byte sectors */
```

one 4 KiB direct write (real trace)

```
block_bio_queue: 251,0 WS 800 + 8 [dd]
# dev 251,0 Write Sync sector 800 8 sectors
```

|      |   |      |
|------|---|------|
| page | 0 | 4096 |
|------|---|------|

page (2)                      offset (1)                      len (1)  
the one bio\_vec of that bio

1 byte 409600 = sector 800 ( $\div 512$ ); 4096 bytes = 8 sectors. The device address is always in sectors

2 the memory side is a list of (page, offset, length): the pages need not be next to each other, like readv/writew's iovec

Book: Love, LKD Ch. 14 (the bio structure)

# blk-mq: bios become requests in queues

1

**merge:** a bio next to a queued request joins it (sectors 800–807 + 808–815 → one request)

2

**schedule:** the I/O scheduler (mq-deadline) may reorder and delay to group I/O

3

**tag:** each request gets a tag 0...depth-1 in the **tag set** (we asked for depth 16)

4

**dispatch:** blk-mq calls the driver's **queue\_rq** for each request

**"mq" = multi-queue:** modern SSDs (NVMe) have many hardware queues, one per CPU. blk-mq gives each CPU its own software queue so CPUs do not fight over one lock. Our driver has **one** hardware queue (blk\_mq\_alloc\_sq\_tag\_set: "sq" = single queue).

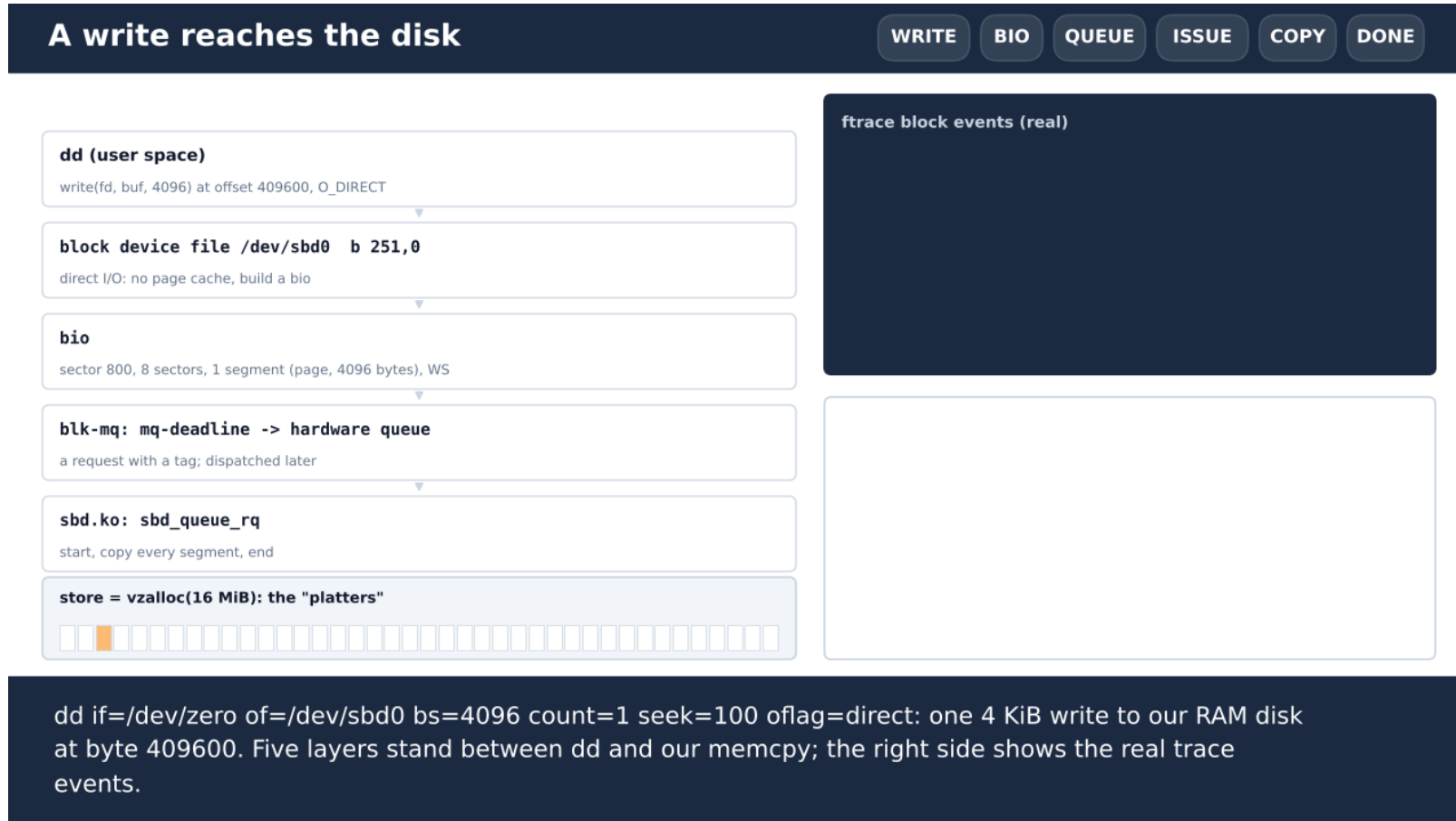
```
● ● ● /sys/block/sbd0/queue (real)
```

```
$ cat scheduler
none [mq-deadline]
$ cat logical_block_size nr_requests
512
32
$ cat max_sectors_kb rotational
127
1
```

**rotational 1?** We never told the kernel our disk is RAM, so it assumes a spinning disk. A real driver sets its limits and features (Homework 15.2 discusses the scheduler).

*Book: Love, LKD Ch. 14 (request queues, I/O schedulers)*

# Watch one 4 KiB write reach the disk



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

## Watch for

1. Byte 409600 is **sector 800**.
2. The **bio**: one 4 KiB piece.
3. **rq\_insert** into mq-deadline.
4. **kworker/0:1H** calls `queue_rq`.
5. **memcpy**, then complete.

## PART 3

# The sbd driver

A RAM disk in about 100 lines: no read, no write, one queue\_rq.

# sbd\_init: memory, a major, a tag set, a disk

● ● ● sbd.c: init (real, core lines)

```
store = vzalloc((size_t)size_mb << 20); /* the "platters" */
major = register_blkdev(0, "sbd"); /* 0: any free major */
err = blk_mq_alloc_sq_tag_set(&tag_set, &sbd_mq_ops,
                             16, SBD_TAG_FLAGS); /* 1 queue, 16 deep */
disk = sbd_alloc_disk(&tag_set); /* blk_mq_alloc_disk */
if (IS_ERR(disk)) { err = PTR_ERR(disk); goto undo_tags; }
disk->major = major; disk->first_minor = 0; disk->minors = 1;
disk->fops = &sbd_fops; /* .owner only */
snprintf(disk->disk_name, DISK_NAME_LEN, "sbd0");
set_capacity(disk, (sector_t)size_mb << (20 - SECTOR_SHIFT));
err = add_disk(disk); /* now /dev/sbd0 exists */
if (err) goto undo_disk;
pr_info("sbd: /dev/sbd0 ready, %d MiB, major %d\n", ...);
```

● ● ● QEMU guest (real)

```
$ insmod sbd.ko size_mb=16
sbd: /dev/sbd0 ready, 16 MiB, major 251
$ cat /sys/block/sbd0/size
32768
$ blockdev --getsize64 /dev/sbd0
16777216
```

**Capacity is in sectors:** 16 MiB = 16 << 20 bytes = 16 << 11 sectors = **32768**, exactly what sysfs shows.

**add\_disk is the moment of truth:** after it, udev, the partition scanner and anyone may send I/O at once. Everything must be ready before.

Book: LDD3 Ch. 16 (registration; old API)

# sbd\_queue\_rq: the whole driver in one function

● ● ● sbd.c (real)

```
static blk_status_t sbd_queue_rq(struct blk_mq_hw_ctx *hctx,
                                const struct blk_mq_queue_data *bd)
{
    struct request *rq = bd->rq;
    loff_t pos = (loff_t)blk_rq_pos(rq) << SECTOR_SHIFT;
    struct req_iterator iter;    struct bio_vec bv;
    blk_mq_start_request(rq);
    if (pos + blk_rq_bytes(rq) > (loff_t)size_mb << 20) {
        blk_mq_end_request(rq, BLK_STS_IOERR); return BLK_STS_OK; }
    rq_for_each_segment(bv, rq, iter) {
        void *buf = bvec_kmap_local(&bv);
        if (rq_data_dir(rq) == WRITE) memcpy(store + pos, buf, bv.bv_len);
        else                          memcpy(buf, store + pos, bv.bv_len);
        kunmap_local(buf);    pos += bv.bv_len;
    }
    blk_mq_end_request(rq, BLK_STS_OK);
    return BLK_STS_OK;
}
```

● ● ● ftrace function\_graph (real)

```
sbd_queue_rq [sbd]() {
    blk_mq_start_request() {
        ktime_get(); __rq_qos_issue();
        blk_add_timer();
    }
    blk_mq_end_request() {
        blk_update_request();
        ... blk_mq_free_request();
    }
}
```

- 1 **start:** the request is ours, timer on
- 2 **copy** every segment
- 3 **end:** status to the block layer, wake the waiter

# A real disk driver finishes later: virtio\_blk

```
● ● ● include/linux/blk-mq.h (real)

532 struct blk_mq_ops {
536     blk_status_t (*queue_rq)(struct blk_mq_hw_ctx *,
        const struct blk_mq_queue_data *);
    ... commit_rqs, complete, timeout, init_hctx ...
};
```

```
● ● ● drivers/block/virtio_blk.c (real lines)

427 static blk_status_t virtio_queue_rq(...)
439     status = virtblk_prep_rq(...); /* start_request */
444     err = virtblk_add_req(vblk->vqs[qid].vq, vbr);
    ... return BLK_STS_OK; /* NOT finished yet! */
351 static void virtblk_done(struct virtqueue *vq)
    /* the interrupt handler, later */
367     blk_mq_complete_request(req);
```

|                    | sbd (RAM)            | virtio_blk, a real disk               |
|--------------------|----------------------|---------------------------------------|
| queue_rq does      | the whole copy       | puts the request in the device's ring |
| data moved by      | the CPU: memcpy      | the device: DMA                       |
| completion         | at once, in queue_rq | later, in the interrupt handler       |
| requests in flight | never more than 1    | up to the queue depth                 |

**Lecture 13 link:** the device raises an interrupt when it is done; the handler completes the request and wakes the process, which slept in state D meanwhile (Lecture 8.1).

## PART 4

# Watching I/O

ext2 on our disk, and the tools that show what the driver did.

# ext2 on /dev/sbd0, seen from the driver (real)

● ● ● mke2fs, sbd trace on (real)

```
$ mke2fs -q /dev/sbd0; dmesg | grep sbd: | tail -4
sbd: WRITE sector 496, 28672 bytes
sbd: WRITE sector 16384, 126976 bytes
sbd: WRITE sector 16632, 126976 bytes
sbd: WRITE sector 16880, 16384 bytes
$ mount -t ext2 /dev/sbd0 /mnt
/dev/sbd0 on /mnt type ext2 (rw,relatime)
```

● ● ● cat after echo 3 > drop\_caches (real)

```
$ cat /mnt/hello.txt
hello on a RAM disk
sbd: READ sector 10, 1024 bytes
sbd: READ sector 12, 1024 bytes
sbd: READ sector 14, 31744 bytes
sbd: READ sector 548, 1024 bytes
```

1

126976 bytes = **248 sectors**: many 4 KiB page writes **merged** into one request

2

$16632 - 16384 = 248$ : the next request starts where the last one ended

3

reading one small file costs **4 requests**: ext2 metadata (inodes, directory) and the file's data

4

ext2 uses **1 KiB** blocks here: 1024-byte requests = one block

**Lecture 12.2 link:** block b of ext2 is sector 2b here. Your ext2cat did the same reads with pread.

# /sys/block/sbd0/stat: the driver's I/O counters (real)

```
cat /sys/block/sbd0/stat after mke2fs, mount, write, cold read (real)
```

```
152  61  1254  16  12  134  1102  26  0  20  43  0 ...
```

| field                                | value       | meaning (Documentation/block/stat.rst)         |
|--------------------------------------|-------------|------------------------------------------------|
| read I/Os                            | 152         | read requests completed                        |
| read merges                          | 61          | read bios merged into a queued request         |
| read sectors                         | 1254        | sectors read: $1254 \times 512 = 642048$ bytes |
| read ticks                           | 16          | milliseconds spent waiting for reads           |
| write I/Os                           | 12          | write requests completed                       |
| write merges                         | 134         | write bios merged: mke2fs's pages              |
| write sectors                        | 1102        | sectors written                                |
| in_flight / io_ticks / time_in_queue | 0 / 20 / 43 | now queued; ms busy; ms waited in total        |

**Tools read this file:** iostat (sysstat) and vmstat compute MB/s and utilisation from exactly these counters, sampled twice. The driver did not count anything itself: blk\_mq\_end\_request did.

# Block trace events: four stops of one request (real)

```
events/block/{bio_queue, rq_insert, rq_issue, rq_complete} one 4 KiB O_DIRECT write (real)
dd-131 [000] 46.329937: block_bio_queue: 251,0 WS 800 + 8 [dd]
dd-131 [000] 46.330943: block_rq_insert: 251,0 WS 4096 () 800 + 8 [dd]
kworker/0:1H-43 [000] 46.331668: block_rq_issue: 251,0 WS 4096 () 800 + 8 [kworker/0:1H]
kworker/0:1H-43 [000] 46.331821: block_rq_complete: 251,0 WS () 800 + 8 [0]
```

| event       | who     | meaning                             |
|-------------|---------|-------------------------------------|
| bio_queue   | dd      | a bio enters the block layer        |
| rq_insert   | dd      | the request goes into the scheduler |
| rq_issue    | kworker | blk-mq hands it to queue_rq         |
| rq_complete | kworker | the driver ended it; [0] = no error |

1 insert → issue: **0.7 ms** waiting in the scheduler

2 issue → complete: **0.15 ms**: our memcpy

3 dd ran 16 MiB with O\_DIRECT in **0.03 s** in total

**Same tools as Lectures 12–13:** tracefs, an event per step, timestamps in seconds since boot. blktrace and iostat are built on these events and counters.

## LAB 15.2

# sbd.c: your own disk

A tag set, a gendisk and a segment loop; then format it and mount it.

# Lab 15.2 tasks, step by step

1 `cd lab15_2; make` builds `sbd_starter.ko`

2 **TODO 2** tag set and disk: `blk_mq_alloc_sq_tag_set`, `sbd_alloc_disk`, fill major, minors, fops, name

3 **TODO 3** `set_capacity` in sectors, `add_disk`; jump to the right `undo` label on error

4 **TODO 1** `queue_rq`: `rq_for_each_segment`, `bvec_kmap_local`, `memcpy` in the right direction

5 test: `mke2fs -q /dev/sbd0; mount /dev/sbd0 /mnt`, write a file, `umount`, mount again, read it

6 `echo 1 > /sys/module/sbd_starter/parameters/trace` and watch your requests in `dmesg`

● ● ● the starter, before TODO 2–3 (real)

```
$ insmod sbd_starter.ko; echo status $?
insmod: can't insert 'sbd_starter.ko': No such device
status 19
$ dmesg | grep sbd | tail -1
sbd: no disk yet: TODO 2 and 3
```

● ● ● the finished lab (real)

```
$ insmod sbd.ko
$ mke2fs -q /dev/sbd0 && mount /dev/sbd0 /mnt
$ echo hi > /mnt/f && cat /mnt/f && umount /mnt
Filesystem label= ...
4096 inodes, 16384 blocks
2 block groups
hi
```

# Checkpoint: test yourself

Q1

A program writes byte 409600 of /dev/sbd0. Which sector does the driver see?

→ sector 800 (409600 / 512)

Q2

Why did 8 writes of 512 bytes reach sbd as one request?

→ they filled one page in the page cache; sync wrote it once

Q3

Who calls sbd\_queue\_rq with mq-deadline on?

→ a kblockd worker (kworker/0:1H), not dd

Q4

set\_capacity(disk, 16 << 20) for 16 MiB. What is wrong?

→ capacity is in sectors: 16 << 11

Q5

A real disk driver's queue\_rq returns before the data moved. Who ends the request?

→ its interrupt handler: blk\_mq\_complete\_request

# Key ideas from today



Block devices are arrays of 512-byte sectors, used through the page cache and file systems.



I/O travels as bios (sector + page list), merged into requests, queued and scheduled by blk-mq.



A blk-mq driver: tag set + gendisk + capacity + add\_disk, and queue\_rq: start, move data, end.



Trace events, /sys/block/.../stat and the driver's log show what really reached the disk.

# Homework 15.2 (due before Lecture 15.3)

## 1. Bad sector

Add a parameter `bad_sector`: requests that include it end with `BLK_STS_IOERR`. With `size_mb=8 bad_sector=1000`, dd 4 KiB blocks with `skip=124` and `skip=125 (iflag=direct)`. Explain both results and the `dmesg` line.

## 2. stat

From the `stat` line on slide 18 compute the average read request size and the bytes written. Why are write merges (134) larger than write I/Os (12)?

## 3. Last close

Explain the trap of slide 7: why did the first try (no fd 3) show the write at once, and both reads? Use `block/bdev.c` lines 634–640 and 666–669.

## 4. 126976

Why were `mke2fs`'s big requests 126976 bytes, not 128 KiB? Use `max_sectors_kb` and the page size.

## 5. Scheduler

`echo none > /sys/block/sbd0/queue/scheduler` and trace the 4 KiB direct write again. Which event disappears, and who runs `queue_rq` now?

# Next lecture and reading

## NEXT: LECTURE 15.3

### USB device drivers

Descriptors, interfaces and endpoints; how usbcore enumerates a device and matches a driver; URBs and interrupt transfers. Lab: a driver for a USB keyboard. The last lecture of the course.

#### Driver book

Corbet, Rubini & Kroah-Hartman, **LDD3** (free), Ch. 16 today (ideas only); Ch. 13 (USB drivers) for 15.3: still close to today's API.

#### Kernel side

R. Love, **LKD**, Ch. 14 (the block I/O layer: bios, request queues, I/O schedulers).

#### Source

Linux 6.8: include/linux/blk-mq.h, blk\_types.h, bvec.h, block/blk-mq.c, block/bdev.c, drivers/block/virtio\_blk.c, Documentation/block/stat.rst (Linux v6.8 source online: [elixir.bootlin.com/linux/v6.8](https://elixir.bootlin.com/linux/v6.8)).

#### Lab files

`lab15_files.zip`: lab15\_2/ (sbd\_starter.c, Makefile), guest/run\_guest.sh.