SSD cont & Block Storage

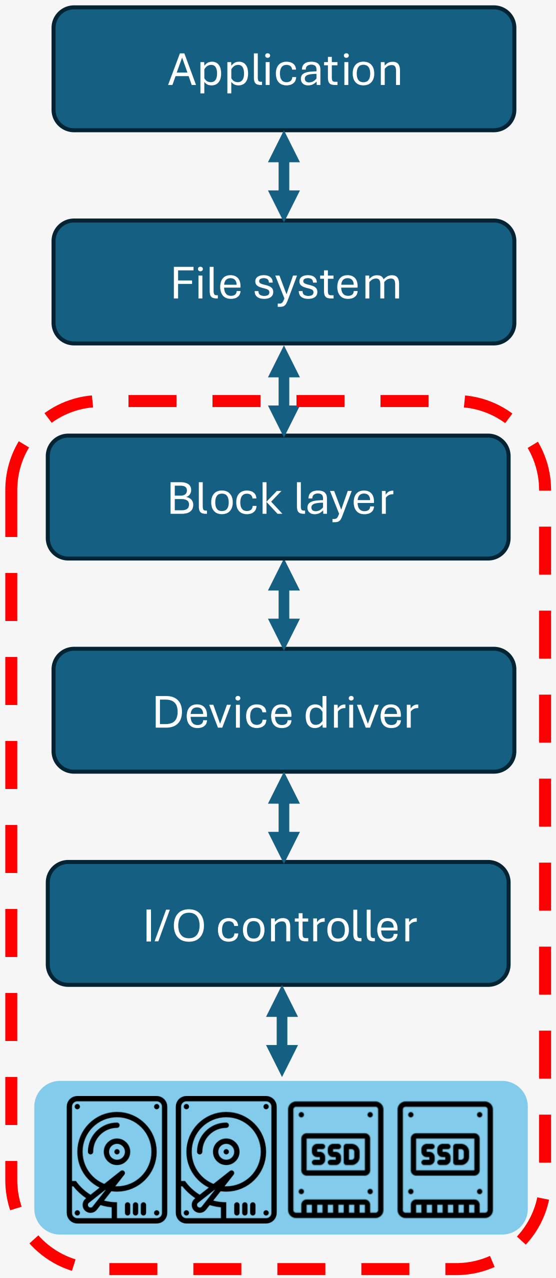
Juncheng Yang

Harvard John A. Paulson School of Engineering and Applied Sciences

HARVARD UNIVERSITY MADSYS
Measurement and Design of Systems Lab

Agenda

- Block layer
- Device driver and I/O controller
- Storage protocol: SATA and NVMe



Three key questions

- What is the role of block layer? What does the interface look like?
- If you are building a large-scale system and need to emulate failures, can you create a flaky disk for testing? How?
- Why do we need NVMe interface, how is it better than SATA?

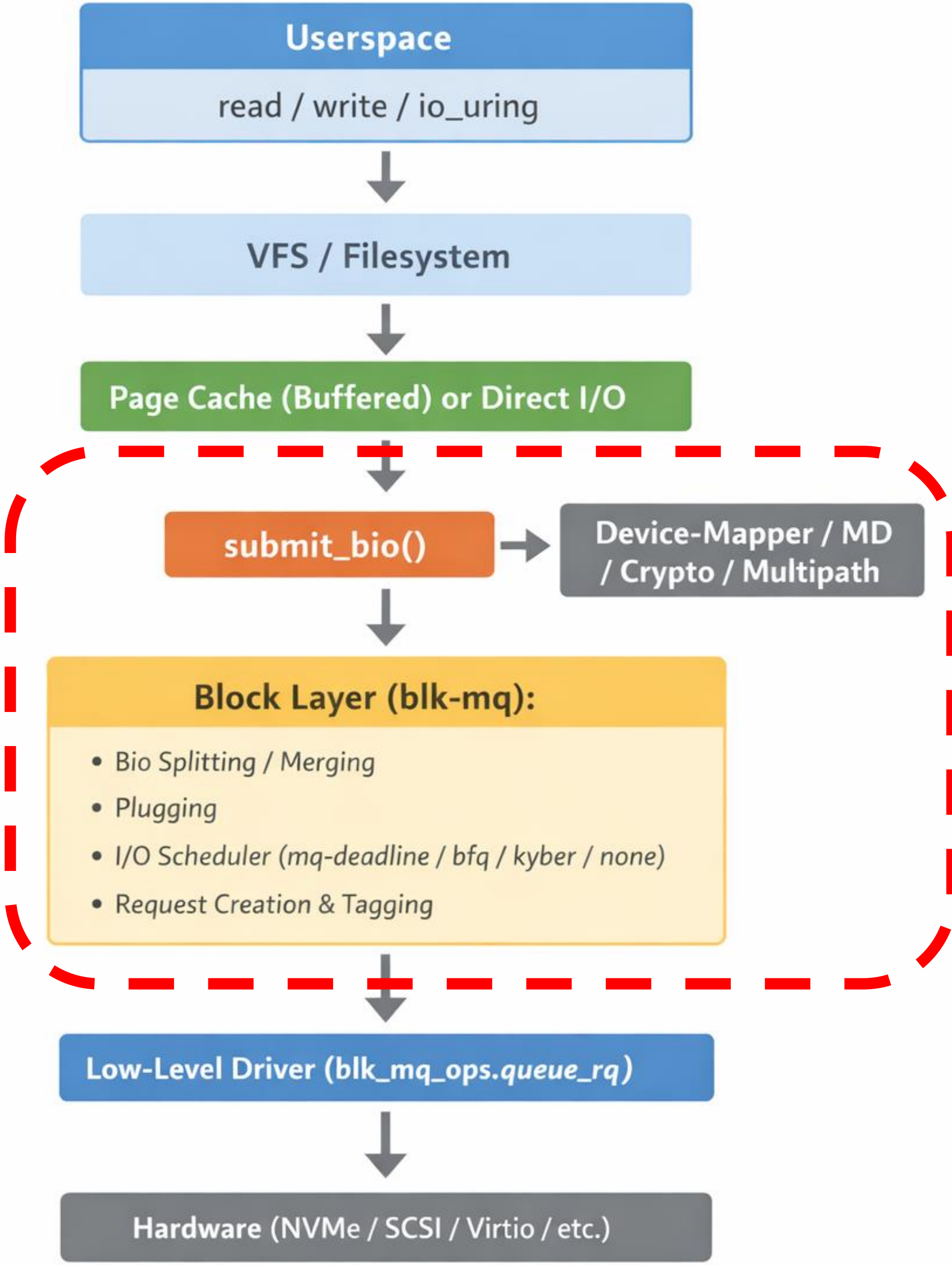
Block layer

Block layer: air traffic controller

- Sit between high-level file systems and low-level drivers / device
- Two roles
  - an abstraction and unified interface: logical block device
    - different hardware: HDD, SSD
    - composed and virtualized device: RAID, remote storage…
    - volume management, partitioning, slicing, mirroring, integrity checking
  - request management
    - I/O representation: setting up bio structure
    - I/O lifecycle management: request submission, dispatch, routing…
    - I/O transformation: translation, split, merge
    - I/O queueing and multiprocessor scaling
    - I/O scheduling: priority, fairness, performance
    - I/O accounting, control and observability

Block layer I/O flow

- File systems: submit a `bio` request
- Block layer
  - split
  - merge
  - batch
  - schedule (re-order)
  - generate request to driver (output)



User-space interface

- Block device is exposed as special files under **/dev**
  - e.g. /dev/sda, /dev/nvme0n1
- Operations (on /dev/<blockdev>)
  - `open()` / `close()`
  - `read()` / `write()` / `lseek()`
  - `ioctl()` for control, geometry, block size, flushing, discard, etc.
    - higher-level tooling uses sysfs and udev: `/sys/block/<dev>/`, `/sys/class/block/<dev>/`
    - /dev/ - Device Nodes (Data)
    - /sys/ - Device Attributes (Metadata)

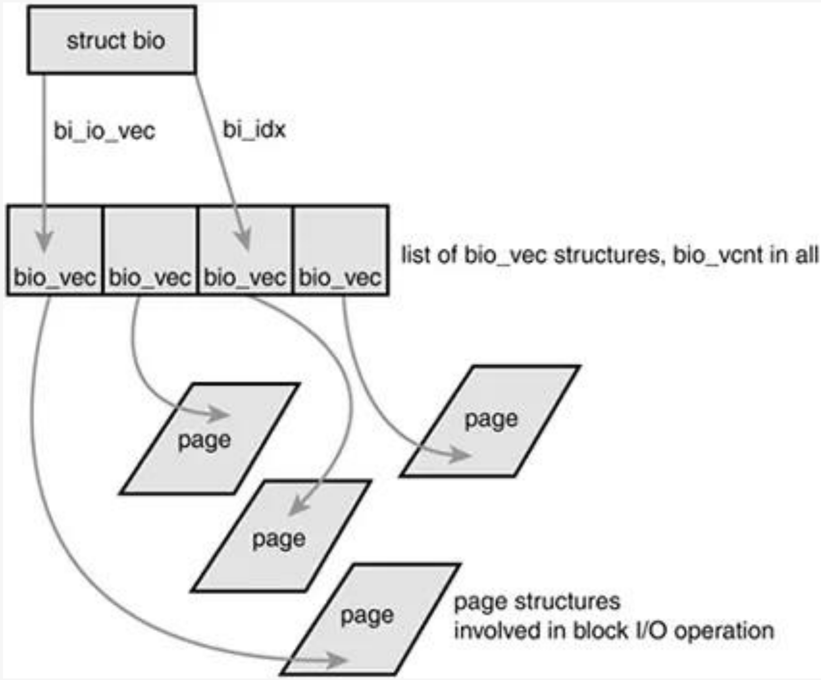
Comparing Block Layer API vs. POSIX APIs

- POSIX APIs
  - user-space facing *file* and *process* interfaces
  - operate on paths, files, directories…
- Block layer operations
  - kernel/driver facing interfaces
  - operate on fixed-size blocks addressed by logical block address (LBA) on a block device
- Different responsibilities for user
  - block: manage layout, consistency, crash recovery
  - block is useful for application that
    - manages their own layout and caching, e.g., database
    - low-level function: imaging, recovery, partitioning

Comparing Block Layer API vs. POSIX APIs

- Key differences
  - Naming
    - POSIX: Locate data via path → inode → file offsets
    - block layer: Locate data via (device, LBA, length)
  - Semantics and guarantees
    - POSIX: more semantic information, block: primitives
  - Concurrency control
    - POSIX: file locking, block: more about request scheduling
  - Caching
  - Error model
    - POSIX: in terms of file, e.g., permission, quota
    - block layer: about device

Kernel interface: `struct bio`



- Interface provided by block layer to file system
- A request to read/write specific memory pages to specific addresses on a block device

```
struct bio {
    struct bio          *bi_next;   // List of requests
    struct block_device *bi_bdev;   // Target device
    unsigned short       bi_flags;  // Read/Write, Sync/Async
    struct bvec_iter     bi_iter;   // Iterator for current position
    struct bio_vec      *bi_io_vec; // The array of memory pages
    // ...
};
```

Kernel interface: operations

- Primary Data Transfer Operations
  - `REQ_OP_READ`, `REQ_OP_WRITE`, `REQ_OP_FLUSH`
- Storage Space Management (Discard/Trim)
  - `REQ_OP_DISCARD`, `REQ_OP_SECURE_ERASE`
- Optimized Write Patterns
  - `REQ_OP_WRITE_ZEROES`, `REQ_OP_WRITE_SAME`
- Zoned Storage Operations (SMR HDDs & ZNS SSDs)

Kernel interface: `struct request`

- Dispatch unit to hardware
- A wrapper that groups one or more `bio` structures together
  - merge and batch to reduce overhead
- **`struct bio`**
  - represents a "Block I/O" unit, points to memory pages and a destination on disk
  - belongs to the **Submitter** (filesystem)
- **`struct request`**
  - represents a "Device I/O" unit
  - managed by the **I/O Scheduler** and the **Block Driver**

Kernel interface: from `bio` to `request`

- Entry: VFS calls `submit_bio()`
- Checks limits: device read-only? partition valid?
- Split: If too large for the hardware, split into smaller bios
- Plug/Merge: add to current request's list, try to merge with previous bio

`bio`

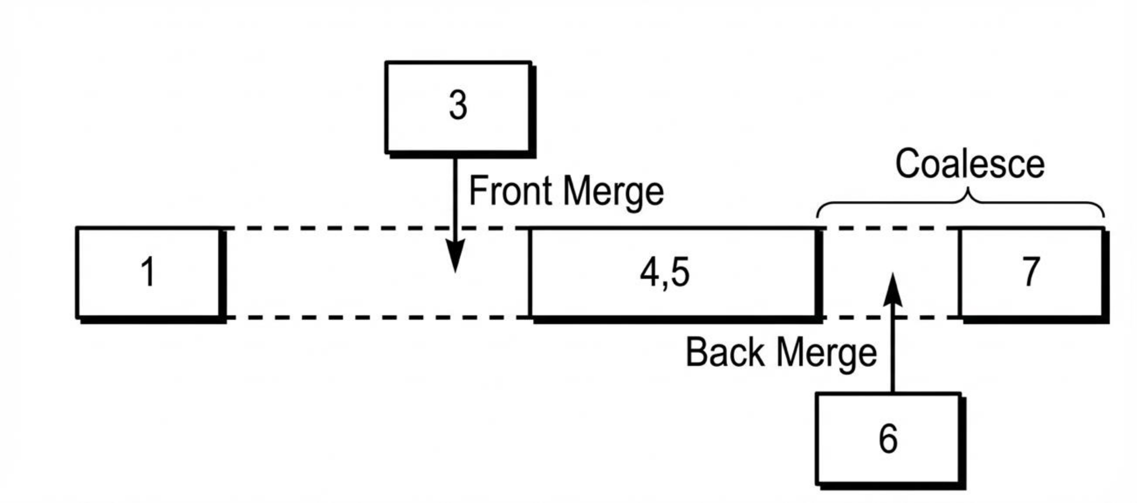
- Scheduler: scheduler sorts/prioritizes request
- Dispatch: send to device driver

`request`

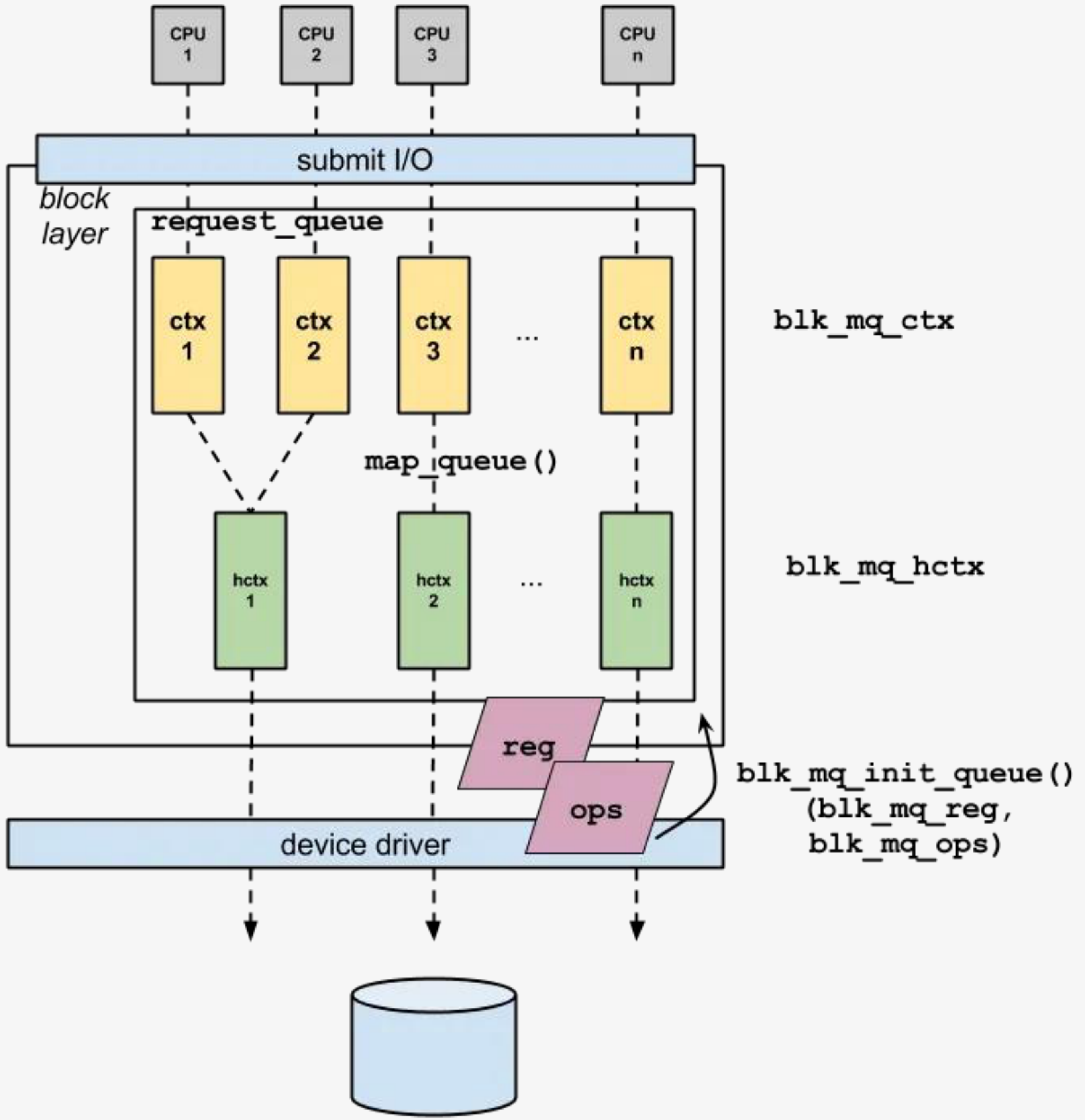
Block layer I/O queueing (Linux)

- The Old World: Single Queue
  - before v3.13 (~2014)
  - designed for HDDs (100 IOPS)
  - lock contention for modern NVMe drives (million IOPS)

Block layer I/O queueing



- The New World: Multi-Queue (`blk-mq`)
  - two levels: software queue and hardware queue
  - software queue (`struct blk_mq_ctx`)
    - per-CPU and no locking, bio is sent to local queue
    - I/O plugging (merging)
  - hardware queue (`struct blk_mq_hctx`)
    - per device and device specific (NVMe drive: 8-256)
    - one or more software queues map to one hardware queue
    - no I/O merge in hardware queue


CPU 1
CPU 2
CPU 3
CPU n
submit I/O
block layer
request_queue
ctx 1
ctx 2
ctx 3
...
ctx n
blk_mq_ctx
map_queue()
hctx 1
hctx 2
...
hctx n
blk_mq_hctx
reg
ops
blk_mq_init_queue()
(blk_mq_reg,
blk_mq_ops)
device driver

Block layer I/O scheduling

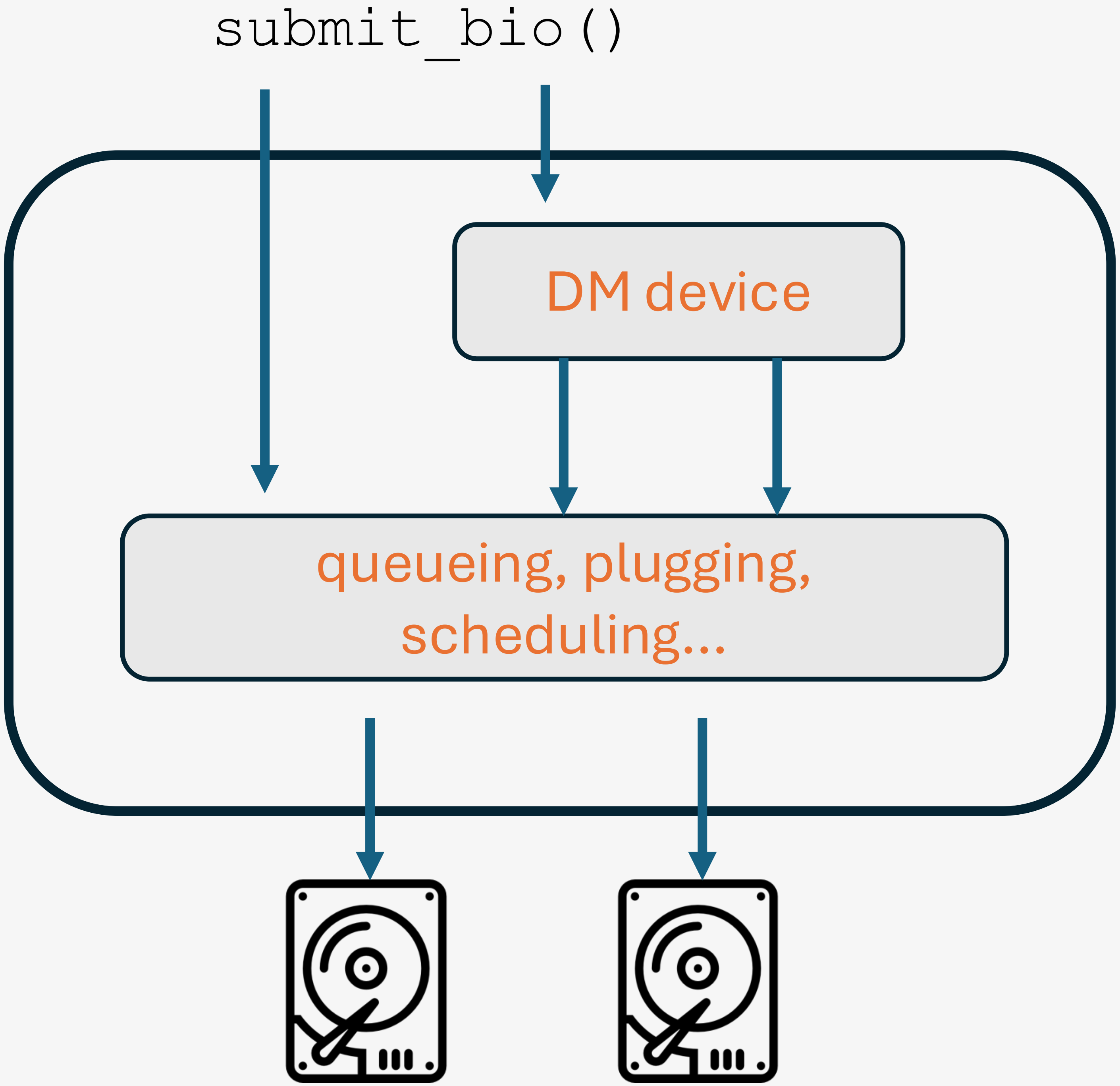
- Goal: re-order `request` for better performance
- Past: schedulers acted like an elevator
    - request data sector 10 and then sector 10,000, scheduler would pick up sector 500 or 1,000 on the way if they come in later
- How about today? Guess?
- Today: *block layer I/O scheduling is less important*
    - SSDs have no moving parts and
    - SSDs have massive parallelism (merge is not necessary)
    - FTL also performs scheduling: waste work
    - overhead: SSD data access <10 μs while scheduling could add a few μs

Block layer I/O scheduling

- Do we still need a scheduler today?
  - most of the systems use **None**
  - sometimes yes: fairness and tail latency
- Common schedulers
  - **mq-deadline**: prioritize read requests (interactive)
  - **Budget Fair Queueing (BFQ)**: complex and slow, assign weight to process
  - **Kyber** (from Meta): built for flash, limits incoming request when latency is high
  - **None**: bypass scheduler

Request transformation and remapping

- Perhaps the most powerful part of the block layer
- Enable virtual block device
- Allow one block device to be "built" on top of another
- Device-mapper (DM)
  - Linux kernel framework for building **virtual block devices** on top of other block devices by applying a **mapping table**
  - used to build LVM, dm-crypt, dm-multipath, and other
- Others
  - remap a partition's LBA to disk LBA
  - bad block management (deprecated)



submit_bio()
DM device
queueing, plugging, scheduling...

Device mapper (DM)

- A DM device is a normal block device
  - e.g. /dev/dm-0, and usually a symlink to /dev/mapper/<name>
- Internally it has
  - **mapped_device**: the kernel object representing the /dev/dm-X
  - **dm_table**: the active mapping table
  - **targets**: transformation or routing policy, e.g., linear, crypt, thin, cache
- The mapping table model
  - basically a list of rules of the form: “For logical sector range [start, start+len), use target T with parameters P.”

How I/O flows through DM

- A **bio** submitted to dm device
  - DM looks up which table entry covers the bio’s logical sector range
  - DM calls the target’s map function, which typically does one of:
    - **remap** sectors onto an underlying device (linear/striped)
    - **clone/split** bios
    - **transform** data
- underlying device(s) complete the bio(s)
- DM aggregates completion and completes the original bio

The power of DM

- Composability ("stacking")
  - targets can be layered: LVM LV (dm-linear) → dm-crypt → NVMe
- Atomic table switching
  - support resize/snapshot/repair mappings online
- A stable user-space control plane
  - user-space talks to DM via `ioctls` with `dmsetup` as the low-level admin tool

Common device mapper targets

- **linear / stripe / raid:** map to one or many devices
- **crypt (dm-crypt):** transparent block encryption
- **thin:** thin provisioning + snapshots
- **cache**
- **multipath:** path failover/aggregation
- **verity:** verified read-only block images

Case study: logical volume management (LVM)

- Physical Volume (PV)
  - a disk/partition initialized for LVM
  - LVM writes a label + metadata onto it so it can be tracked and grouped
- Volume Group (VG)
  - a VG is a **pool of storage** created from one or more PVs
- Logical Volume (LV)
  - an LV is what you use: a **virtual block device** allocated from a VG
  - you put filesystems on it, use it as swap, give it to databases, etc.

Case study: logical volume management (LVM)

- Online growth
  - extend VGs by adding PVs, then extend LVs without rebuilding the whole layout
- Snapshots
  - **classic snapshots** (copy-on-write)
  - **thin snapshots** when using thin provisioning
- Thin provisioning (thin pools)
  - thin LVs allocate physical blocks **on demand** from a thin pool
- Caching (dm-cache)
  - LVM can build cache LVs: caches hot blocks on a faster device

Case study: logical volume management (LVM)

- DM is the kernel mapping engine
  - map a virtual block device (LV) to one or more underlying block devices based on a table
- **LVM**: user-space manager + metadata + policy
- **DM**: kernel data path that routes/transforms bio

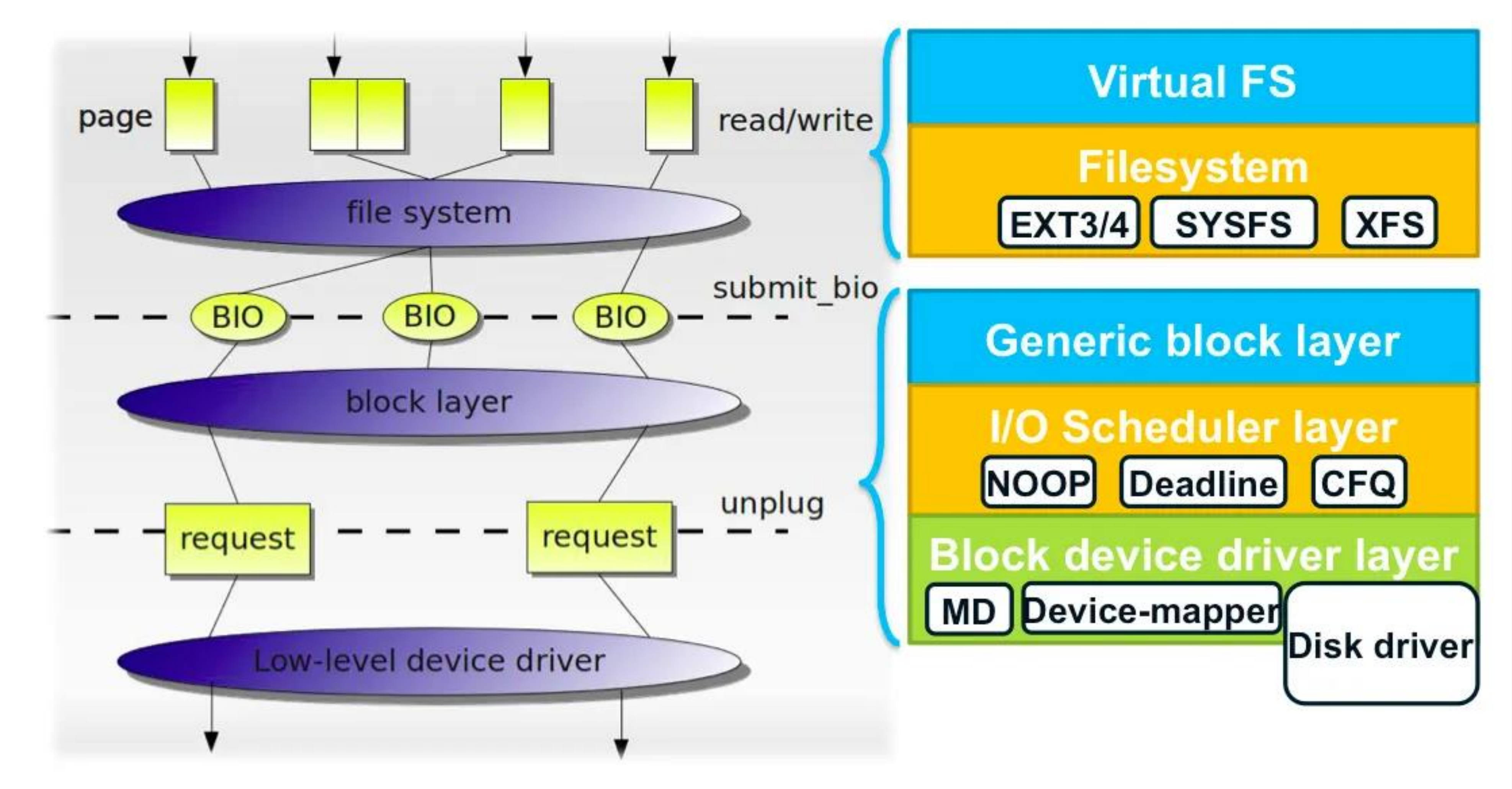
Case study: Loop

- Loop: file-backed block device
  - takes a regular file sitting on an existing filesystem and presents it to the kernel as a block device (/dev/loop0)
- How it works
  - when a bio is submitted, the loop driver translates that block request into a standard file read() or write() operation on the underlying file

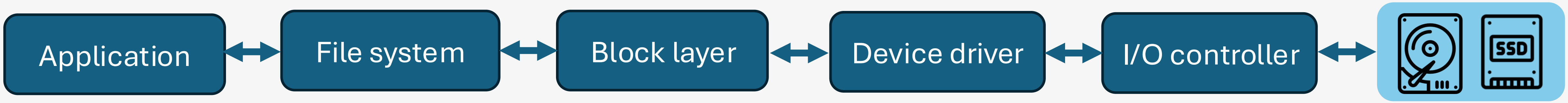
Case study: RAM-based devices

- RAM disk: fast but no persistence
- **brd**
  - carve out a fixed chunk of memory and map as a block device
- **zram**
  - compressed RAM disk
  - often used as a swap device

Block layer summary



Device Driver and I/O Controller



Drive driver

- A piece of **kernel code** that knows the "secret language" of a specific I/O controller
- Role:
  - **binds** to a discovered hardware device
  - **exposes** a block device to the block layer
  - **translates** block-layer requests into hardware commands, then completes them back to the block layer

I/O controller

- **Hardware** that sits between the CPU and the storage medium and implements the command protocol + data movement
- Example
  - NVMe controller: on device
  - SATA/AHCI controller: on motherboard and on device
- Key functions
  - Protocol handling
  - Data transfer
  - Command queuing and execution
  - Error handling
  - Optional buffering

Storage protocol and interface

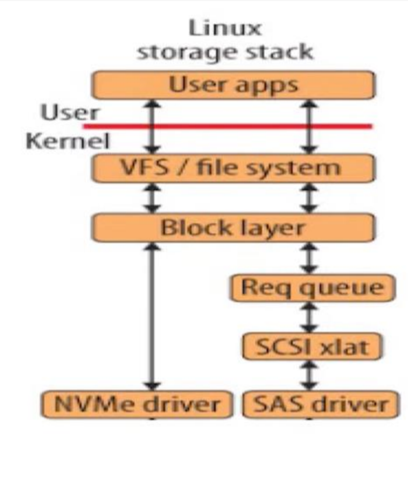
SATA/AHCI interface

- Used by HDDs and some SSDs
- AHCI (Advanced Host Controller Interface) protocol
- Native Command Queuing (NCQ)
  - 32 outstanding commands (allow some limited re-ordering)
  - not enough parallelism to feed data to SSD
- Linux uses a translation layers in device driver layer to make SATA drives pretend to be SCSI drives
  - historical reason: when moving from IDE subsystem to SATA, many features have been implemented for SCSI

SAS (Serial Attached SCSI) protocol

- Used by enterprise HDDs
- Performance
  - SATA: half-duplex, can send OR receive data
  - SAS: full duplex, read and write simultaneously
- Reliability
  - two physical ports for redundancy
- Daisy chain
  - connect hundreds drives in a chain for easy extension

NVMe protocol



- NVMe: non-volatile memory express
- Designed for low-latency and high parallelism over PCIe
  - direct PCIe connection
  - no SATA/SAS controller overhead
  - bypass unnecessary layers
- 64K queues with 64K commands each
  - submission queue (SQ) and completion (CQ)
  - facilitates parallelism
  - placed in host memory
  - can be mapped to core for lock-avoidance and NUMA affinity

NVMe protocol: the setup

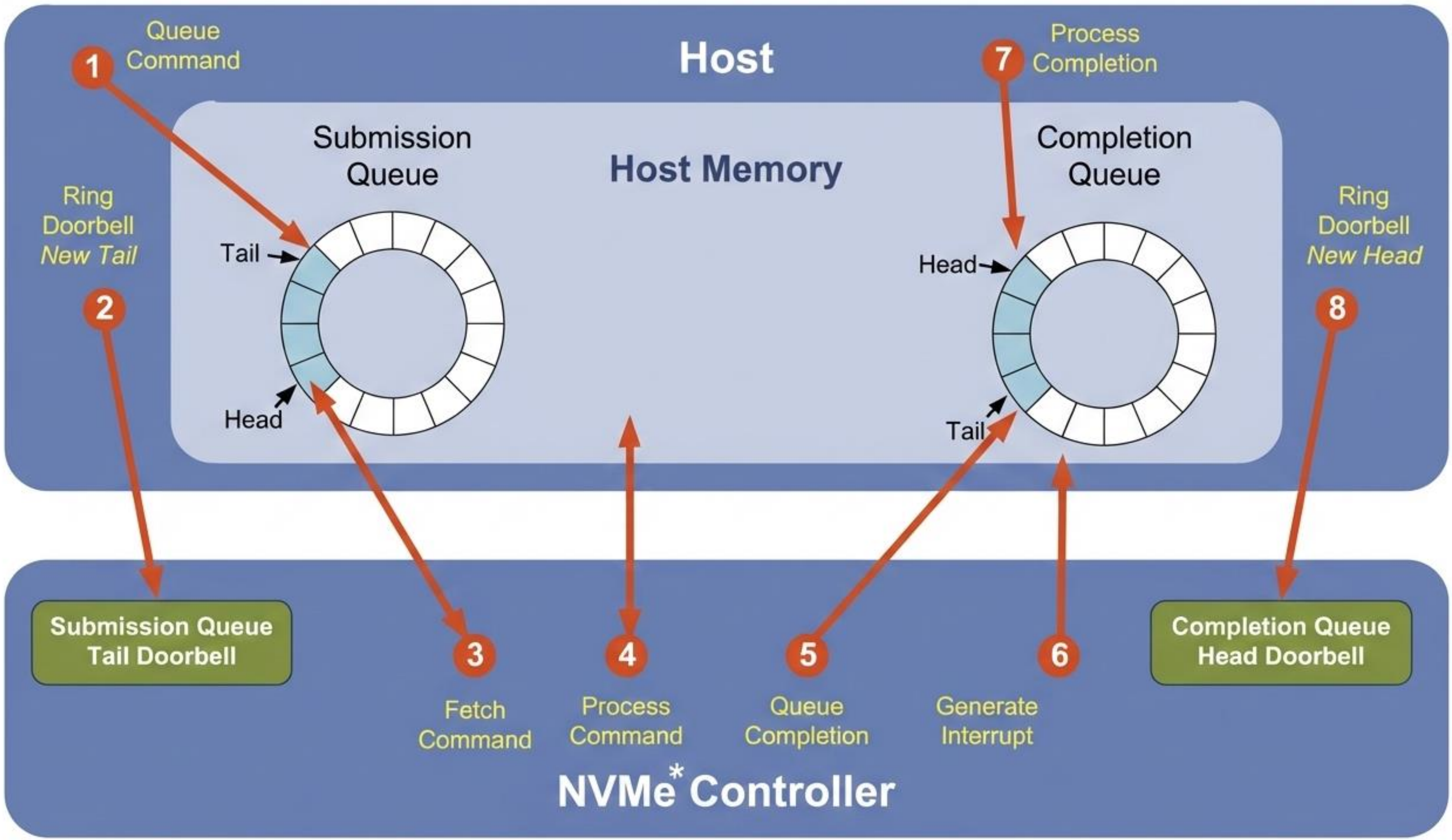
- **The Queue Pair (SQ & CQ):** NVMe communicates using pairs of queues located in **Host RAM**
- **Submission Queue (SQ):** A circular buffer where the Host places commands
- **Completion Queue (CQ):** A circular buffer where the Device places results
- **The Doorbell Registers:** The SSD exposes a small region of high-speed memory (BAR0) mapped into the CPU's address space. This contains the "Doorbell" registers

NVMe protocol: I/O flow

- Host writes to memory (RAM)
  - OS builds a NVMe Command and places it into the next free slot in the Submission Queue (SQ) in DRAM
- Host rings the Doorbell (MMIO)
  - OS writes the new index value to the SSD's SQ Tail Doorbell register to wake up SSD
  - mechanism: PCIe Memory Write (fast because the CPU doesn't wait for response)
- Controller fetches the command (DMA)
  - SSD Controller sees the doorbell register change
  - issues a PCIe Read Request (DMA) to fetch command
  - has the instruction and executes the data transfer (read flash and DMA data to the host buffer)

NVMe protocol: I/O flow

- Once the SSD finishes the read and puts the data in your buffer
- Device writes status to memory (RAM)
  - SSD creates a Completion Entry and DMA to the next free slot of the Completion Queue (CQ) in Host DRAM
- Device interrupts the Host (MSI-X)
- Host processes the completion
- Host rings the CQ Doorbell
  - The OS writes to the SSD's CQ Head Doorbell register indicating it has processed data


Host
Queue Command
1
Process Completion
7
Submission Queue
Host Memory
Completion Queue
Ring Doorbell New Tail
2
Tail
Head
Head
Tail
Ring Doorbell New Head
8
Submission Queue Tail Doorbell
3
4
5
6
Completion Queue Head Doorbell
Fetch Command
Process Command
Queue Completion
Generate Interrupt
NVMe* Controller

NVMe protocol: why faster than SATA?

- Lockless Parallelism
  - each CPU core can have its own private SQ/CQ pair
- Efficient MMIO
  - host only writes to the Doorbell (no wait for response)
  - never reads from the device registers in the critical path (hundreds of cycles)
- Variable Queue Depth
  - NVMe queues can hold up to 64,000 commands (SATA was limited to 32)

NVMe protocol: advanced capabilities

- Namespace management
  - *logical* partitioning of storage
  - noisy neighbors
- Virtualization support
  - efficiently share drive across VMs with direct hardware access, e.g., Single Root input/output virtualization (SR-IOV)
- Sanitize command: secure erasing

NVMe protocol: advanced capabilities

- Power management: up to 32 states (PS0-PS4+)
  - fine-grained control over power consumption vs performance tradeoff
  - PS0: operational, PS4+: deep sleep
  - PS1, PS2: intermediate (throttle performance)
  - PS3: sleep/suspend: stop handling I/O commands, but keep memory refreshed
- Atomic write unit (AWUN)
  - guarantee write size <= AWUN to complete atomically, avoid “torn writes” due to power failure
  - *could* be useful for file systems and databases
- End-to-end data protection
- Reservation and locking: support multi-host use case

NVMe: an evolving standard

Version	Year	Important features
1.0	2011	Initial specification, defining the basic high-performance interface, queuing mechanisms, and end-to-end data protection
1.1	2012	Essential for early enterprise use cases, features include **multipath I/O**, **namespace sharing for** multi-host access to a single namespace
1.2	2014	**NVMe Reservations** for shared namespaces, Host Memory Buffer (**HMB**) for DRAM-less SSDs, and atomic write unit
1.3	2017	**Sanitize command**, **Streams** to hints for efficient data placement, and **Telemetry** logs
1.4	2019	**Asymmetric Namespace Access (ANA)** for optimized multipathing, **Persistent Event Log**, and **Rebuild Assist**
2.0	2021	A major architectural update that modularized the specification. New command sets: **Zoned Namespaces (ZNS)**, **Key Value (KV)**, zand support for HDDs over NVMe interface.
2.1	2024	**Key Per I/O encryption, Flexible Data Placement (FDP)** for host-directed data placement, **NVMe Network Boot**

Open Compute Project (OCP) standard a superset for data center use case

Summary

- Block layer
  - user-space device interface
  - kernel interface
  - IO queueing
  - IO scheduling
  - IO transformation: device mapper
- Storage protocols
  - NVMe I/O request flow

Next time

- File systems

- By end of next week
  - sign up for one paper, TA will send out the signup sheet
  - you can choose the paper you are interested in if you sign up early