CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 21

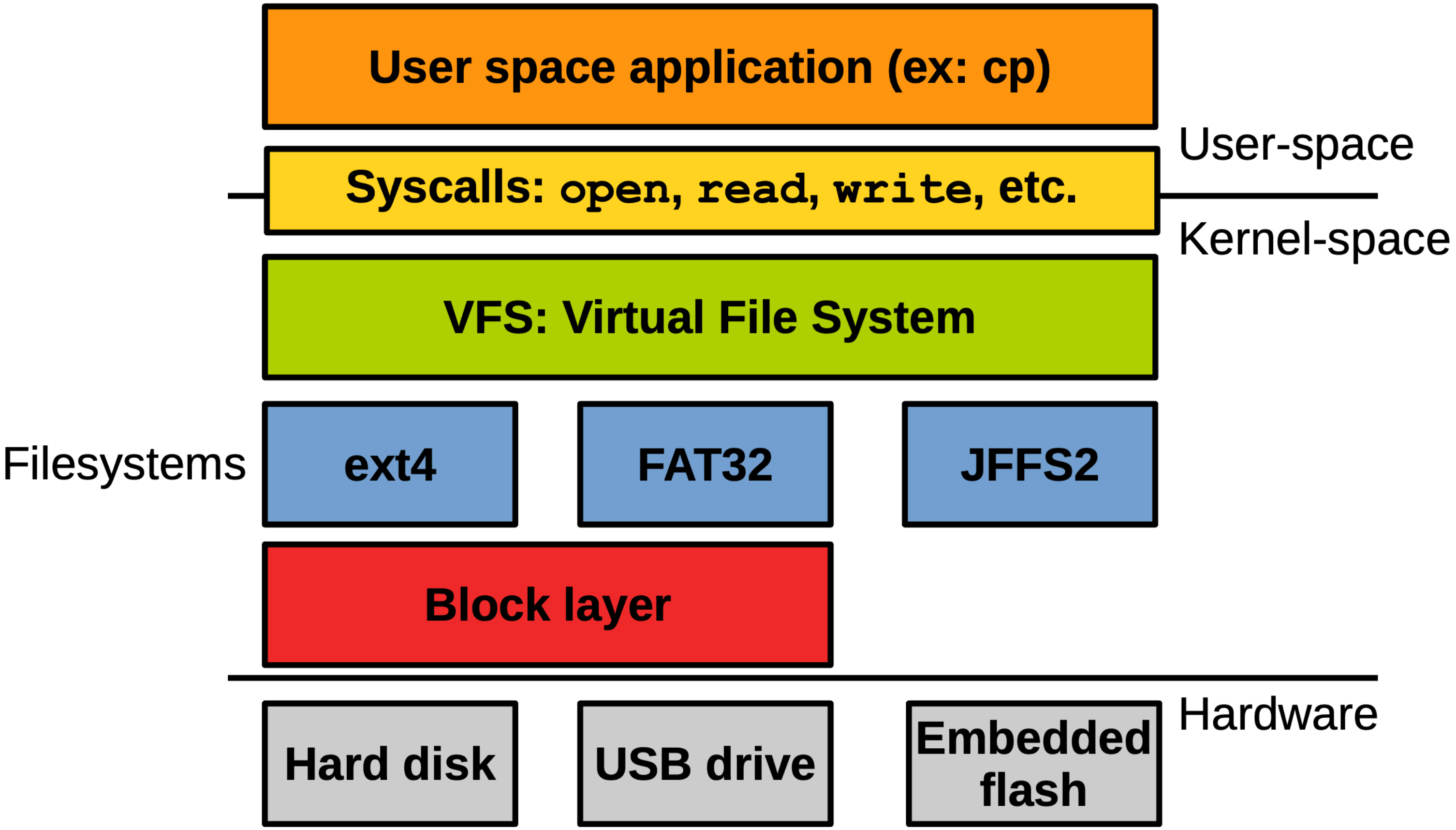
The Block I/O Layer

April 29, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.


User space application (ex: cp)
User-space
Syscalls: open, read, write, etc.
Kernel-space
VFS: Virtual File System
Filesystems
ext4
FAT32
JFFS2
Block layer
Hardware
Hard disk
USB drive
Embedded flash

Layering

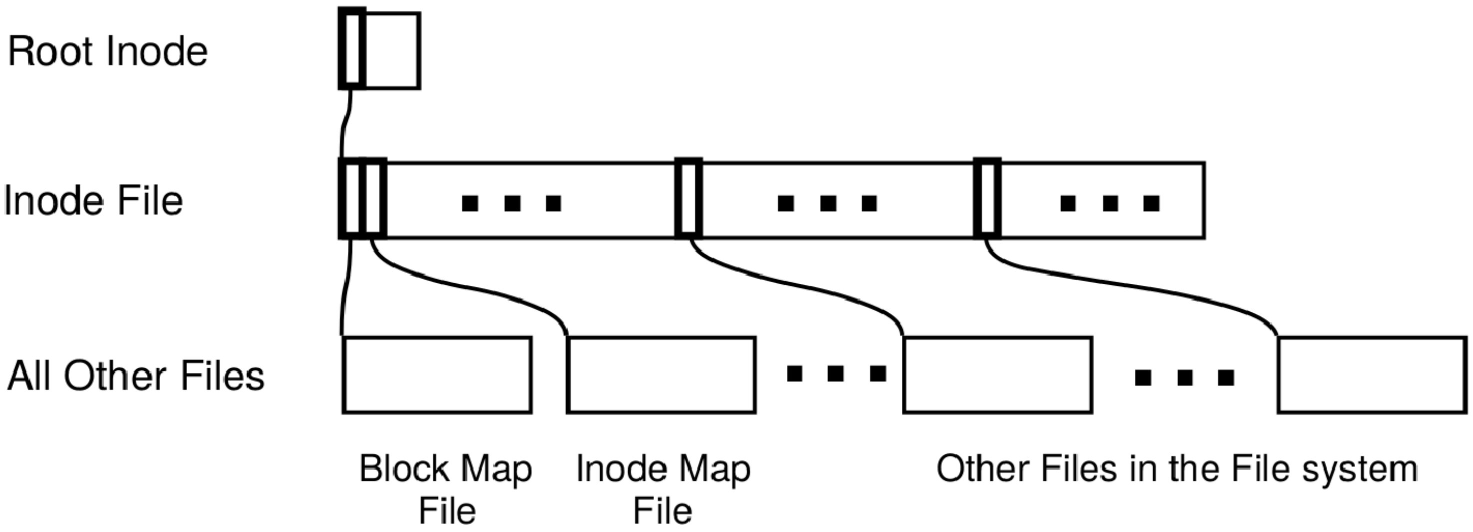
- Human:
  - "Jump to slide #20 of /tmp/slides.ppt" → Random access (/tmp mounted on /dev/sda4)
- Powerpoint application:
  - Convert "slide #20" to byte offset (e.g. 20000-th byte)"
- System calls:
  - open()+lseek()+read() → "read(/tmp/slides.ppt, byte offset 20000)"
- File System:
  - Get the file information of **/tmp/slides.ppt → inode #76** (today)
  - Convert byte offset into block offset in a file (e.g. block offset # 20)
  - Get the block number at the block offset # 20 (e.g. block number 6543), **e.g., using multi-level indexed file → Previous lecture**
  - To block layer: "read logical block number 6543" (logical wrt this partition /dev/hda4)
- Block layer:
  - Converts LBN # 6543 of /dev/hda4 to disk sector#
  - Block layer to device driver: "read sector #"

Copy-on-Write (CoW) File System

- File System Design for an NFS File Server Appliance
  - USENIX Winter 1994
  - Write-Anywhere File Layout (WAFL): the core design of NetApp
- Inspired by LFS
  - Never overwrite a block like LFS
  - No segment cleaning unlike LFS
- Key idea
  - represent a filesystem as a single tree; never overwrite blocks (CoW)

WAFL Layout: A Tree of Blocks

- A root inode: root of everything
- An inode file: contains all inodes
- A block map file: indicate free blocks
- An inode map file: indicates free inodes



Why Keeping Metadata in Files

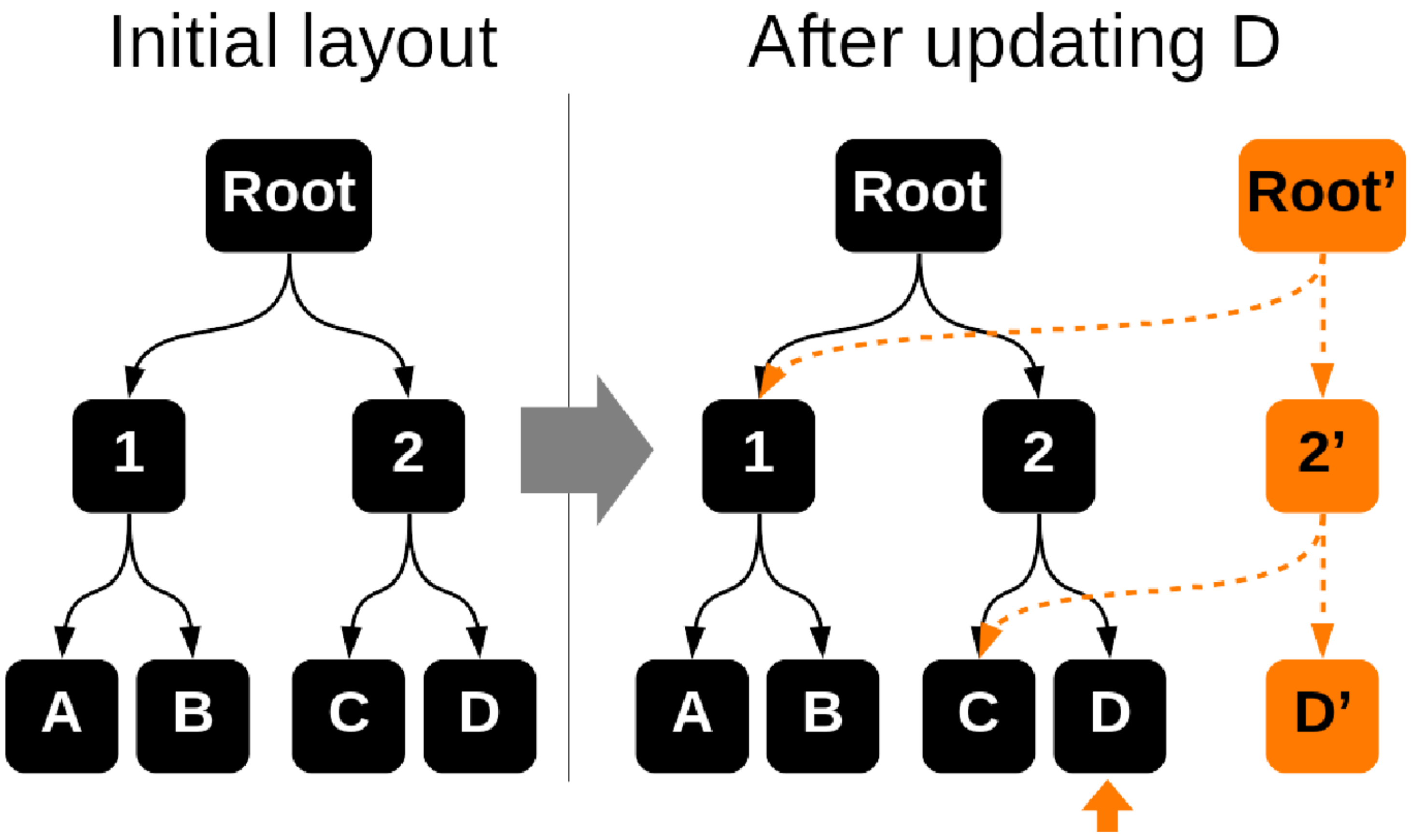
- Allow metadata blocks to be written anywhere on disk
  - This is the origin of “Write Anywhere File Layout”
- Easy to increase the size of the file system dynamically
  - Add a disk can lead to adding i-nodes
- Eanble copy-on-write to create snpshots
  - copy-on-write new data and meatada on new disk locations
  - fixed metadata locations are cumbersome

WAFL Read

- Reads are similar to UFS, once we find the inode for a file
  - root inode → inode file → inode
  - inode: file offset → disk block mapping

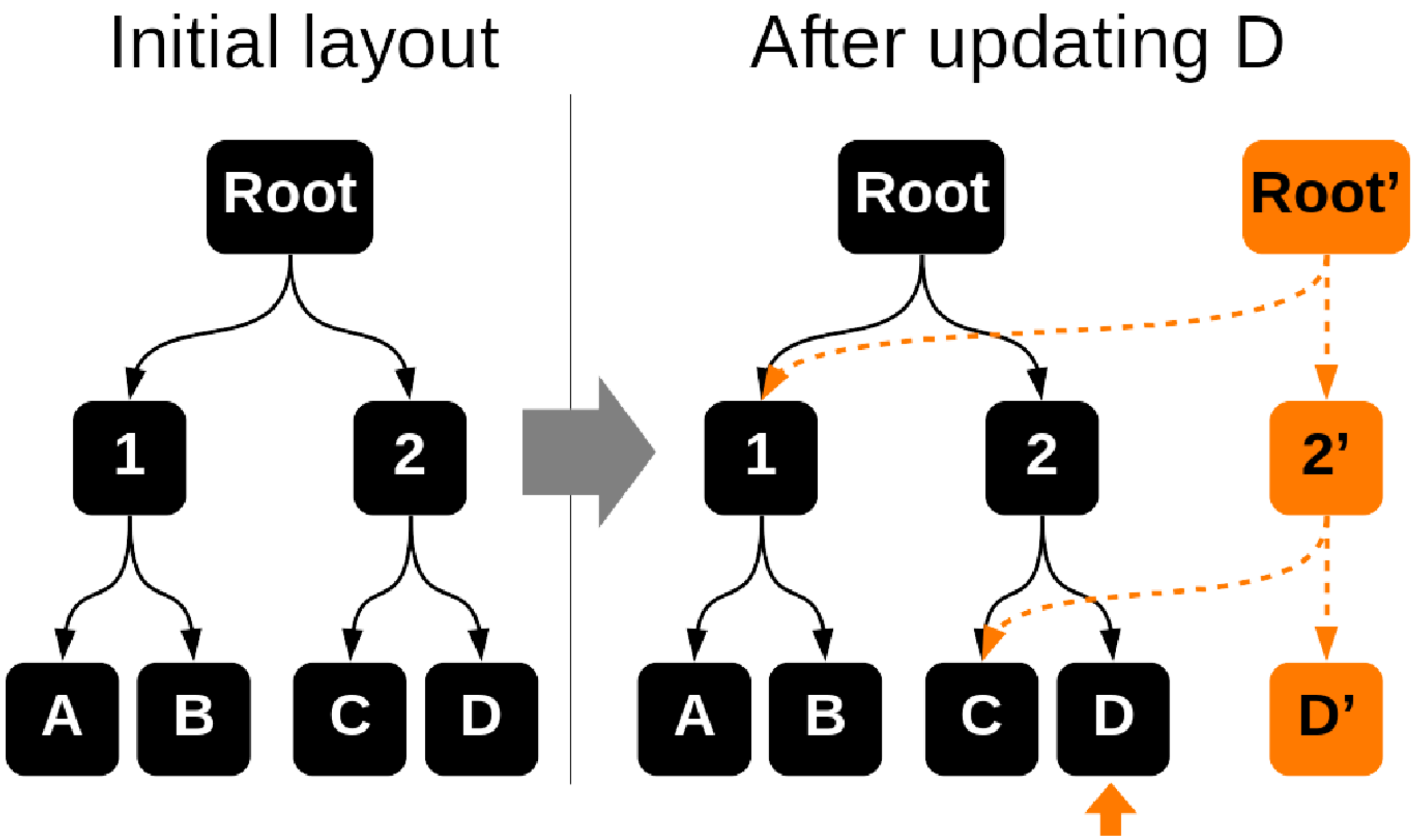
WAFL Write

- WAFL filesystem is a tree of blocks
- Never overwrite blocks → Copy-on-Write



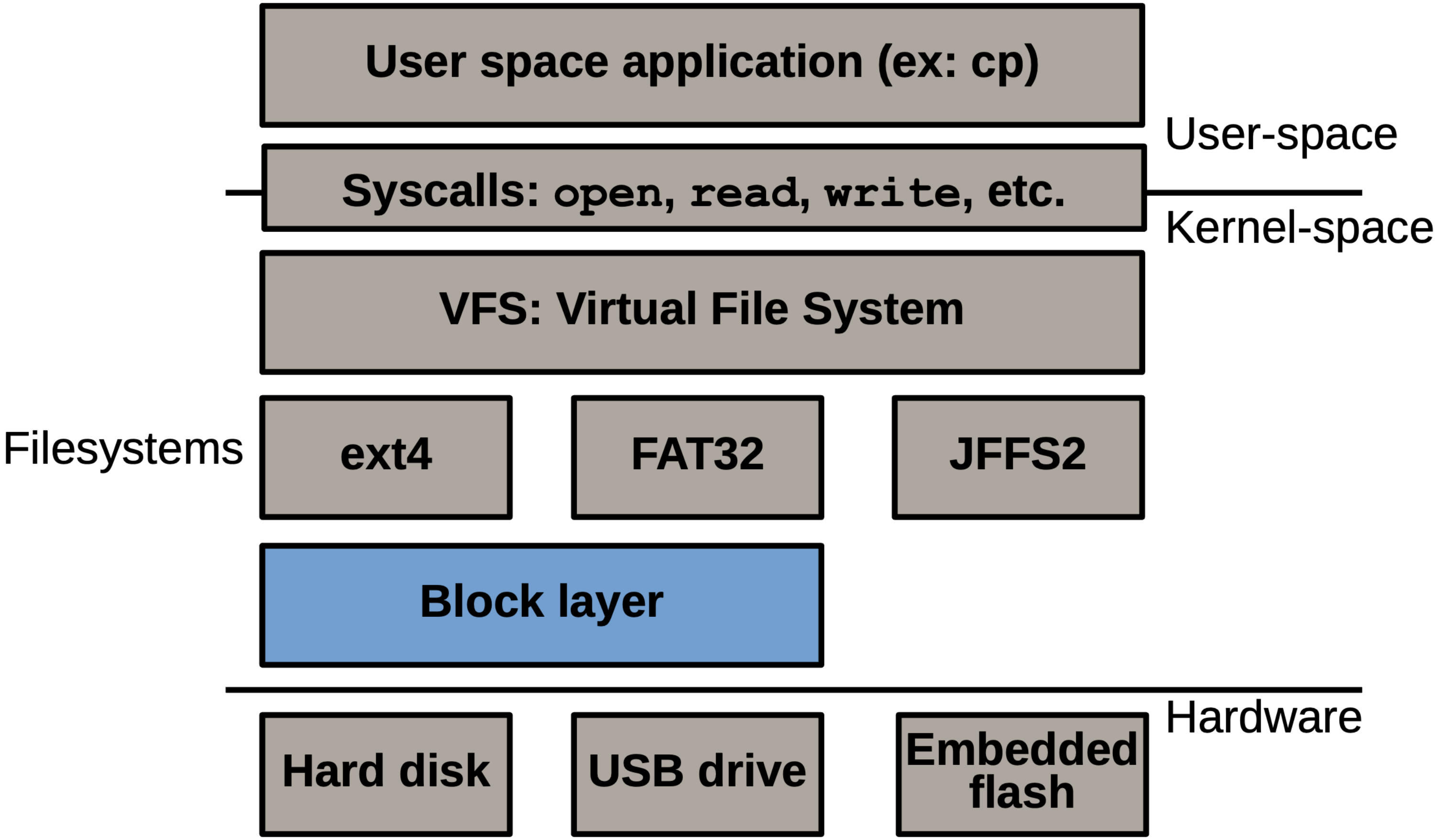
Crash Consistency in LFS

- Each root inode represents a consistent snapshot of a file system



CoW FS in Linux

- btrfs (b-tree file system)
  - A file system is a tree of four CoW-optimized B-trees
- ZFS
  - Default file system of Solaris



User space application (ex: cp)
User-space
Syscalls: open, read, write, etc.
Kernel-space
VFS: Virtual File System
Filesystems
ext4
FAT32
JFFS2
Block layer
Hardware
Hard disk
USB drive
Embedded flash

Block I/O Layer

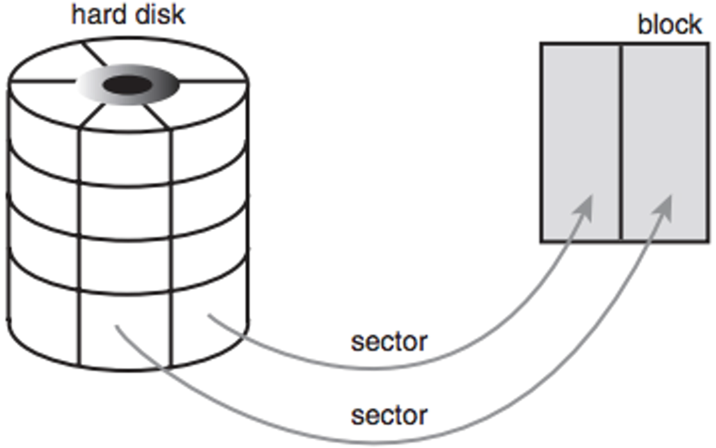
- Block devices and the block layer
- Buffers and buffer heads
- The “bio” structure and request queues
- I/O schedulers

Block Devices and Block Layer

- BIO layer
- Request layer
- I/O scheduler

Anatomy of a Block Device

- Sector
  - Minimum addressable unit in a block device
  - Physical property of the device → hard sector, device block
- Block
  - Unit of filesystem access → filesystem block, I/O block
  - Multiple of a sector (device limitation) and multiple of a page (kernel limitations)
  - Mostly 4KB



mapping from sectors to blocks

Buffers and Buffer Heads

- Buffer: blocks are stored in memory
- Buffer head: metadata of a buffer

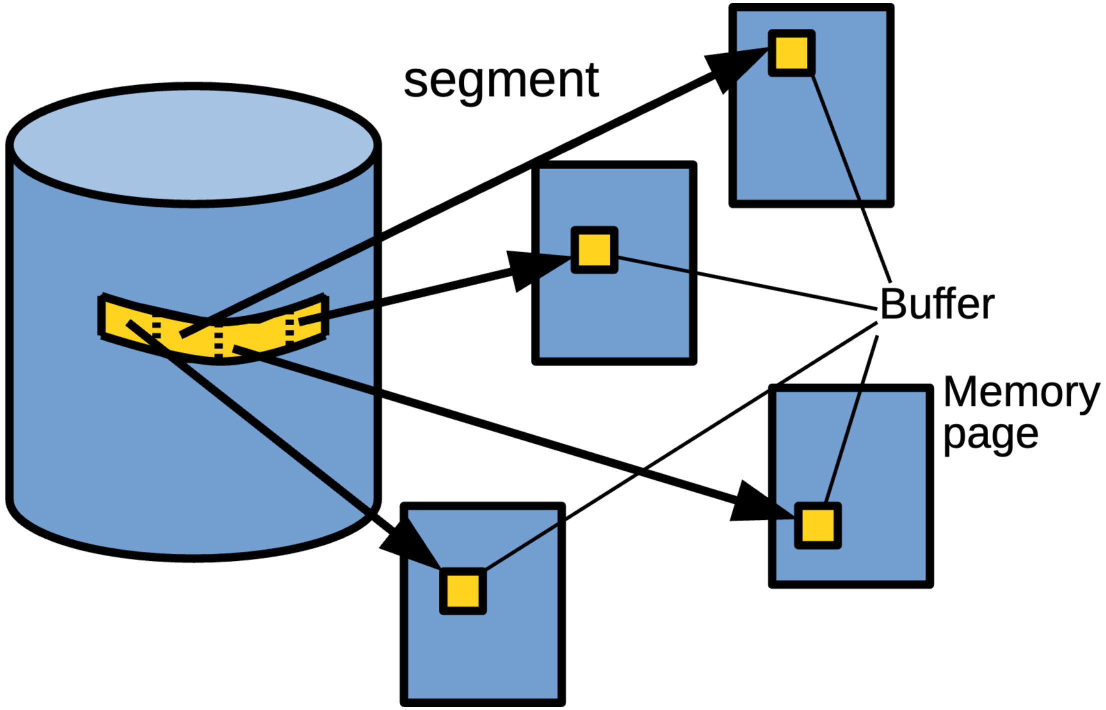
```c
/* linux/include/linux/buffer_head.h */
struct buffer_head {
    unsigned long         b_state;          /* buffer state flags */
    struct buffer_head;   *b_this_page;     /* list of page's buffers */
    struct page           *b_page;          /* associated page */
    sector_t              b_blocknr;        /* starting block number */
    size_t                b_size;           /* size of mapping */
    char                  *b_data;          /* pointer to data within the page */
    struct block_device   *b_bdev;          /* associated block device */
    bh_end_io_t           *b_end_io;        /* I/O completion */
    void                  *b_private;       /* reserved for b_end_io */
    struct list_head      b_assoc_buffers;  /* associated mappings */
    struct address_space  *b_assoc_map;     /* associated address space */
    atomic_t              b_count;          /* use count: get_bh(), put-bh() */
}
```

Buffer State: b_state

```
/* linux/include/linux/buffer_head.h*/
enum bh_state_bits {
    BH_Uptodate,/* Contains valid data */
    BH_Dirty,   /* Is dirty */
    BH_Lock,    /* Is locked */
    BH_Req,     /* Has been submitted for I/O */
    BH_Uptodate_Lock,/* Used by the first bh in a page, to serialise
                      * IO completion of other buffers in the page */
    BH_Mapped,  /* Has a disk mapping */
    BH_New,     /* Disk mapping was newly created by get_block */
    BH_Async_Read,  /* Is under end_buffer_async_read I/O */
    BH_Async_Write, /* Is under end_buffer_async_write I/O */
    BH_Delay,   /* Buffer is not yet allocated on disk */
    BH_Boundary,    /* Block is followed by a discontiguity */
    BH_Write_EIO,   /* I/O error on write */
    BH_Unwritten,   /* Buffer is allocated on disk but not written */
    BH_Quiet,   /* Buffer Error Prinks to be quiet */
    BH_Meta,    /* Buffer contains metadata */
    BH_Prio,    /* Buffer should be submitted with REQ_PRIO */
    BH_Defer_Completion, /* Defer AIO completion to workqueue */

    BH_PrivateStart,/* not a state bit, but the first bit available
                     * for private allocation by other entities */
```

The bio Structure

- Basic container for an active block I/O operation
- An individual buffer being divided into segments, it needs not to be contiguous in memory

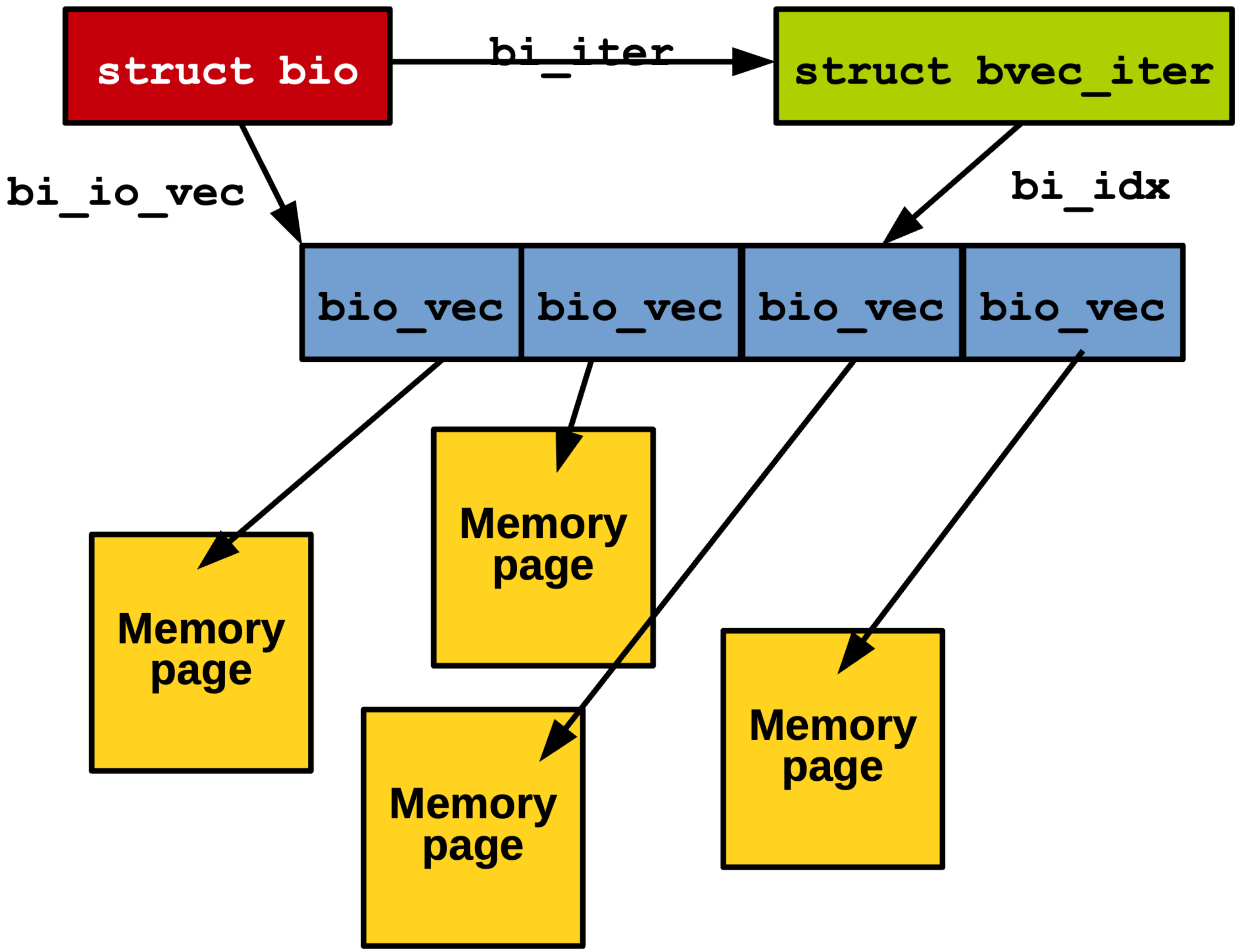


The bio Structure

```c
/* linux/include/linux/blk_types.h */
struct bio {
    struct bio          *bi_next;          /* list of requests */
    struct block_device *bi_bdev;          /* associated block device */
    unsigned short      bi_flags;          /* status and command flags */
    unsigned int        bi_phys_segments;  /* number of segments */
    struct bvec_iter    bi_iter;           /* vector iterator */
    unsigned int        bi_seg_front_size; /* size of front segment */
    unsigned int        bi_seg_back_size;  /* size of last segment */
    bio_end_io_t        *bi_end_io;        /* I/O completion method */
    void                *bi_private;       /* owner private data */
    unsigned short      bi_vcnt;           /* number of bio_vecs */
    unsigned short      bi_max_vecs;       /* maximum bio_vecs possible */
    atomic_t            __bi_cnt;          /* usage counter */
    struct bio_vec      *bi_io_vec;        /* bio_vec list */
    struct bio_vec      bi_inline_vecs[0]; /* inline bio vectors */
    /* ... */
};
```

```c
/* linux/include/linux/bvec.h */
struct bvec_iter {
    sector_t     bi_sector; /* target address on the device in sectors  */
    unsigned int bi_size;   /* I/O count */
    unsigned int bi_idx;    /* current index into bi_io_vec */
    /* ... */
};

/* linux/include/linux/bio.h */
struct bio_vec {
    /* pointer to the target physical page: */
    struct page *bv_page;
    /* length in bytes of the buffer: */
    unsigned int    bv_len;
    /* offset inside the page where the buffer resides: */
    unsigned int    bv_offset;
};
```

The bio Structure



Request Queues

- Block devices maintain request queues to store pending I/O requests
- Request queues are represented by the request_queue structure defined in include/linux/blkdev.h
- Requests are added to the queue by a file system
- Requests are pulled from the queue by the block device driver and submitted to the device

```
struct request_queue {
    /* Together with queue_head for cacheline sharing */
    struct list_head     queue_head;
    struct request       *last_merge;
    struct elevator_queue   *elevator;
    /* ... */
};
```

Request Queues

- A single request:
  - Represented by "struct request"
  - Can operate on multiple consecutive disk blocks, so it consists of one or more bio objects

```
struct request {
    struct list_head queuelist;
    union {
        struct call_single_data csd;
        u64 fifo_time;
    };

    struct request_queue *q;
    struct blk_mq_ctx *mq_ctx;
    /* ... */
};
```

I/O Schedulers

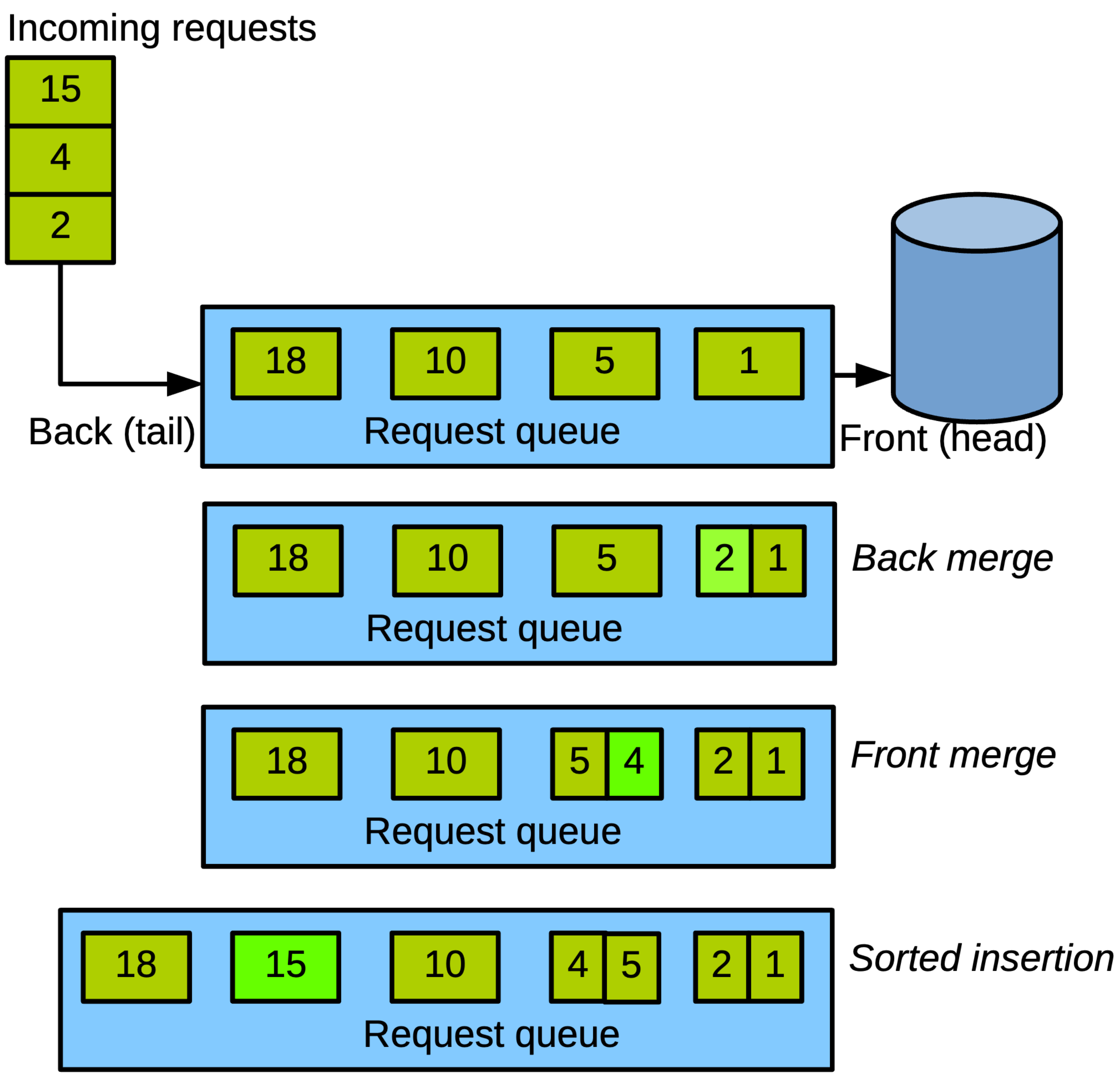
- Directly sending requests to the disk as they arrive is sub-optimal:
  - Increase random accesses
  - The kernel tries to reduce disk seek as much as possible
- The kernel combines and re-order I/O requests in the request queue
  - merging, sorting
- Rules for merging and sorting are defined by the I/O scheduler
  - Multiple I/O scheduler models implemented in Linux
- The I/O scheduler virtualizes the disk as the process scheduler virtualizes the CPU

Linus Elevator

- Default I/O scheduler until v2.4
- Define where an upcoing request should be added into the queue:
  - front merge, back merge
  - sorted insertion
- Goal: minimize disk seek, best global throughput

Linus Elevator

- If a request to an adjacent on-disk sector is in the queue, the existing request and the new request merge into a single request
- If a request in the queue is sufficiently old, the new request is inserted at the tail of the queue to prevent starvation of the other, older, requests
- If a suitable location sector-wise is in the queue, the new request is inserted there. This keeps the queue sorted by physical location on disk.
- Finally, the request is inserted at the tail of the queue.

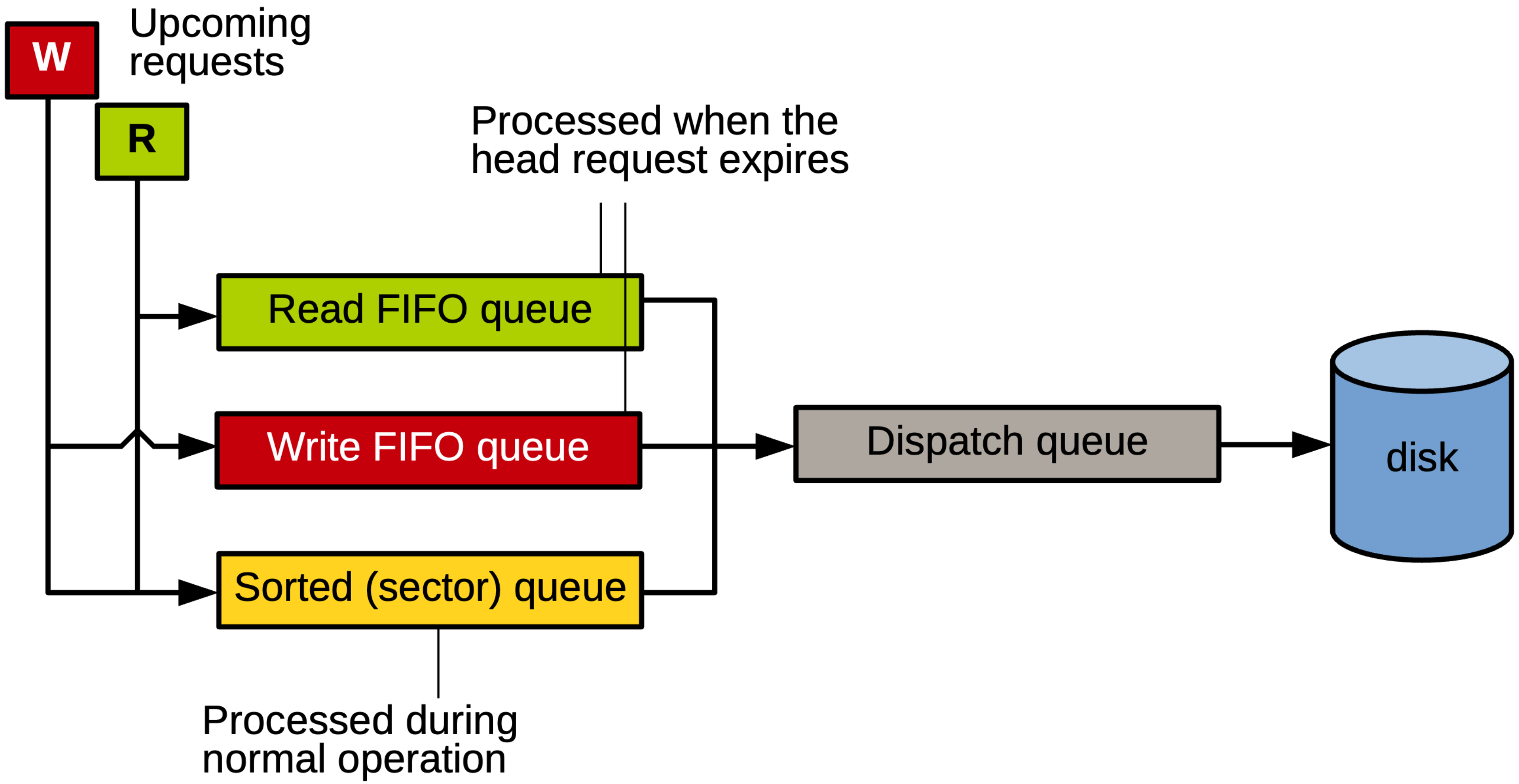

Incoming requests
15
4
2
Back (tail)
18
10
5
1
Request queue
Front (head)
18
10
5
2
1
Request queue
Back merge
18
10
5
4
2
1
Request queue
Front merge
18
15
10
4
5
2
1
Request queue
Sorted insertion

Problems with Linus Elevator

- Goal: minimize disk seek, best global throughput
  - Can cause starvation
- Writes starve reads
  - Buffer I/O operation with buffer page cache
  - Write operations are buffered to page cache → asynchronous
  - Read operations upon page cache miss should be immediately handled → synchronous
  - Read latency is important for the system → read starvation must be minimized

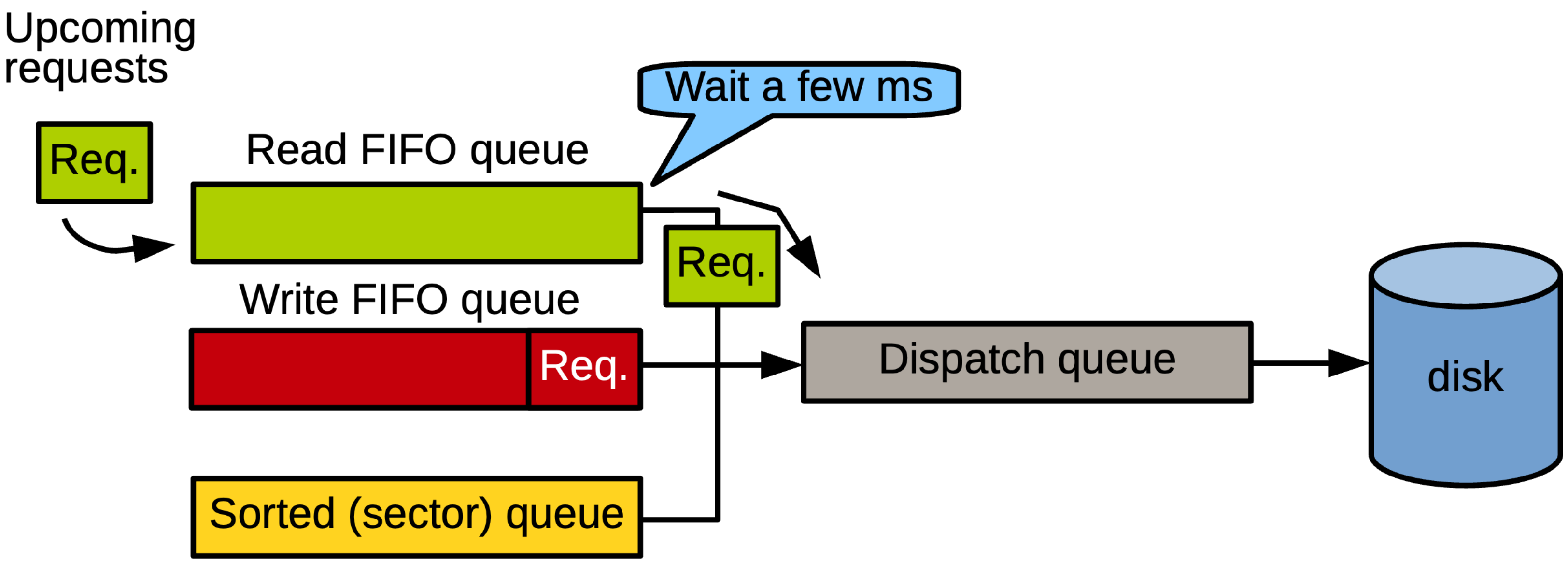
The Deadline I/O Scheduler

- Tries to provide fairness while maximizing the global throughput
- Each request is given an expiration time, the deadline:
  – Reads = now + .5s, Writes = now + 5s



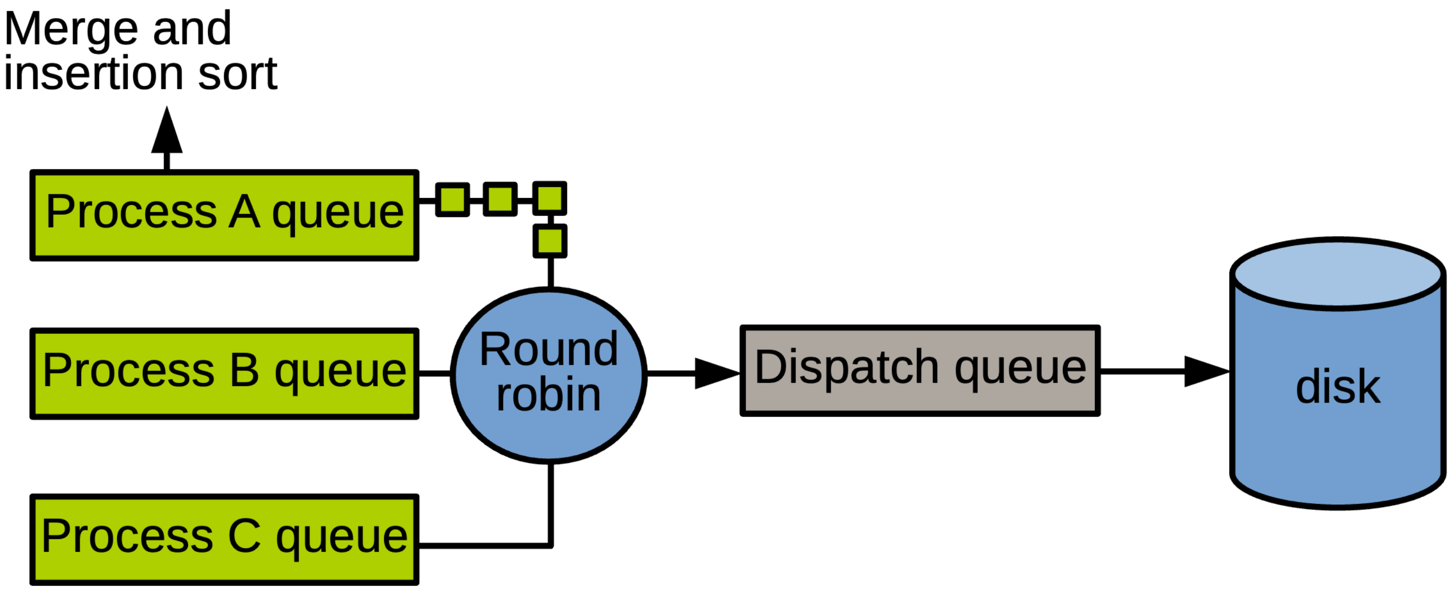
The Anticipatory I/O Scheduler

- Tries to improve the throughput of the deadline scheduler
- Anticipation heuristic
  - Instead of immediately seeking back, it waits for a few milliseconds hoping an application sends other I/O requests.



The Complete Fair Queuing (CFQ)

- Default I/O scheduler (for a long time ...)
- Per-process request queues
- Serves the queues round robin



The NOOP I/O Scheduler

- Does not perform anything in particular apart from merging sequential requests
- Used for truly random devices such as NAND Flash SSDs

Configuring I/O Scheduler

- I/O scheduler can be selected at boot time as a kernel parameter:
  - elevator=<value>-<value> could be either of cfq, deadline, or noop
- Or you can choose an I/O scheduler per device

```
$> cat /sys/block/<block device name>/queue/scheduler
$> echo noop /sys/block/<block device name>/queue/scheduler
```

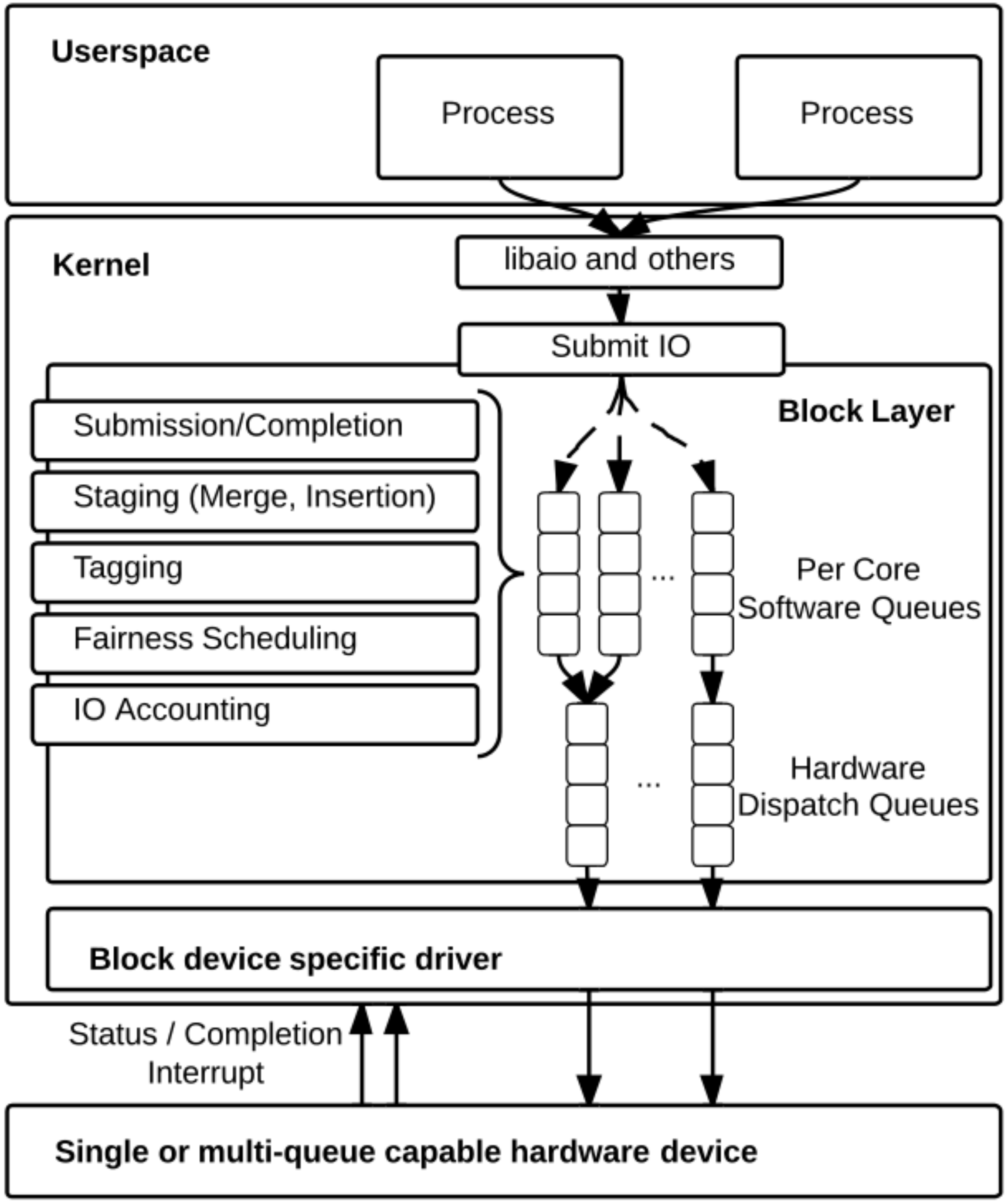
Adding a New I/O Scheduler

```
/* linux/include/linux/elevator.h */
struct elevator_type
{
    /* managed by elevator core */
    struct kmem_cache *icq_cache;

    /* fields provided by elevator implementation */
    union {
        struct elevator_ops sq;
        struct elevator_mq_ops mq;
    } ops;
    size_t icq_size;    /* see iocontext.h */
    size_t icq_align;   /* ditto */
    struct elv_fs_entry *elevator_attrs;
    char elevator_name[ELV_NAME_MAX];
    struct module *elevator_owner;
    bool uses_mq;

    /* managed by elevator core */
    char icq_cache_name[ELV_NAME_MAX + 6];  /* elvname + "_io_cq" */
    struct list_head list;
};
```

```c
/* linux/include/linux/elevator.h */
struct elevator_ops
{
    elevator_merge_fn *elevator_merge_fn;
    elevator_merged_fn *elevator_merged_fn;
    elevator_merge_req_fn *elevator_merge_req_fn;
    elevator_allow_bio_merge_fn *elevator_allow_bio_merge_fn;
    elevator_allow_rq_merge_fn *elevator_allow_rq_merge_fn;
    elevator_bio_merged_fn *elevator_bio_merged_fn;

    elevator_dispatch_fn *elevator_dispatch_fn;
    /* ... */
};
```

Linux blk-mq

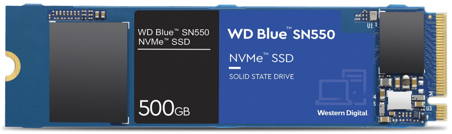
- blk-mq: Multi-Queue Block IO Queueing Mechanism
- Since v3.13
- *"Blk-mq allows for over 15 million IOPS with high-performance flash devices (e.g. PCIe SSDs) on 8-socket servers, though even single and dual socket servers also benefit considerably from blk-mq"*

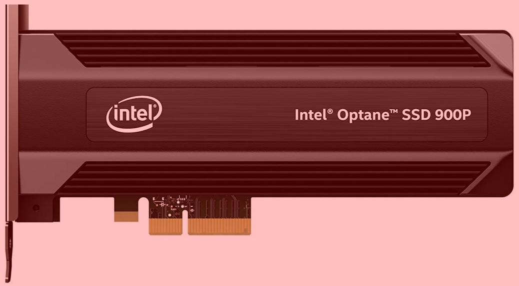


Refer to here.

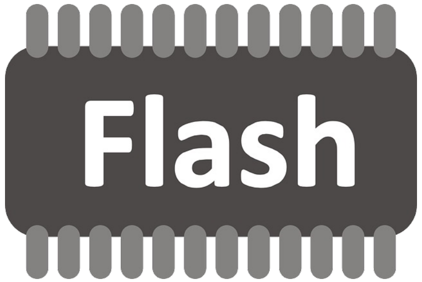
Mainstream SSDs in the NVMe "Hat"

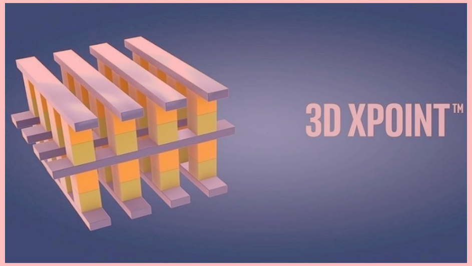
SSD Flavors:





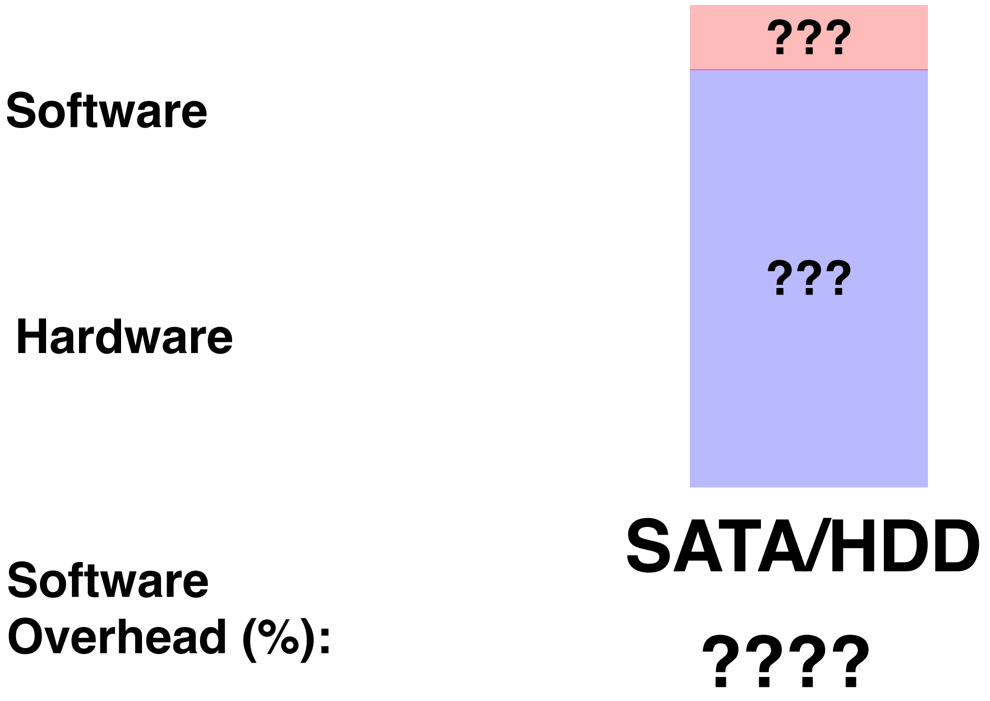
Storage Media:





Why NVMe?

- Performance: reduce the legacy storage stack overhead
  - SATA
  - HDD



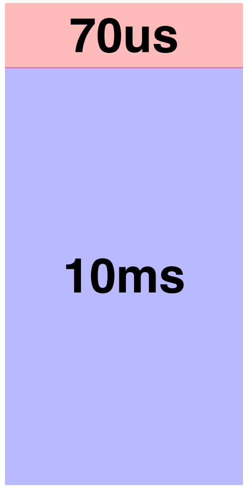
From "Performance Analysis of SAS/SATA and NVMe SSDs": https://www.architecting.it/blog/performance-analysis-sas-sata-nvme/

Why NVMe?

- Performance: reduce the legacy storage stack overhead
  - SATA
  - HDD

Software

Hardware

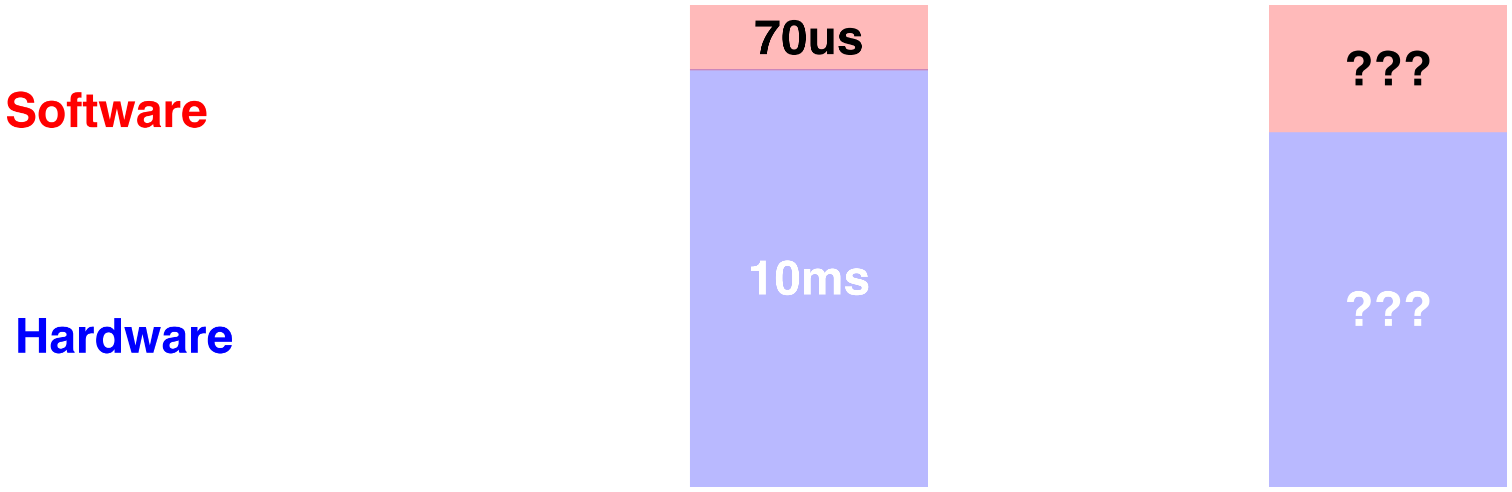


SATA/HDD

Software Overhead (%): **~0.7%**

Why NVMe?

- Performance: reduce the legacy storage stack overhead
  - SATA
  - HDD → SSD

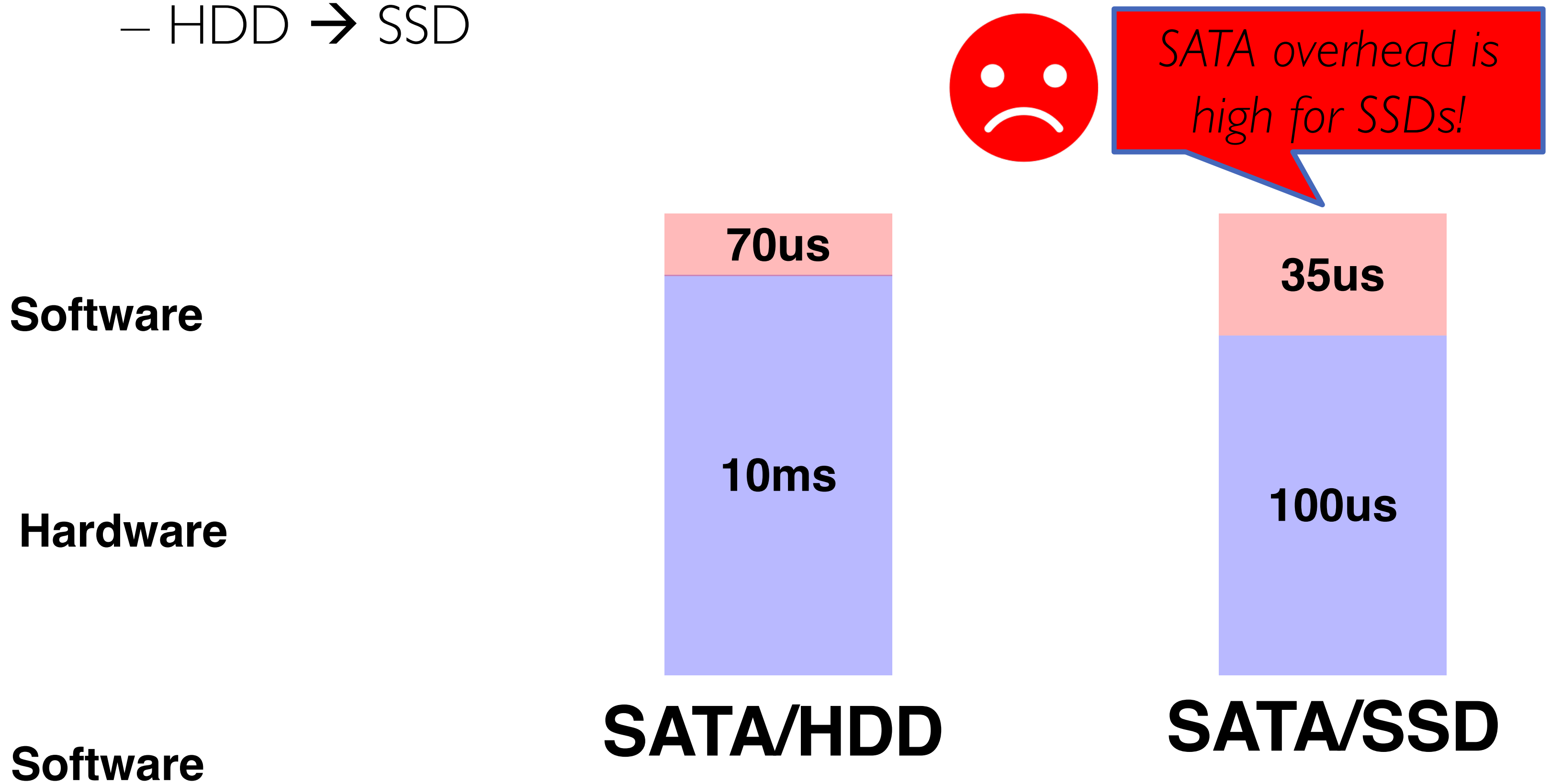


Software Overhead (%):

~0.7%

Why NVMe?

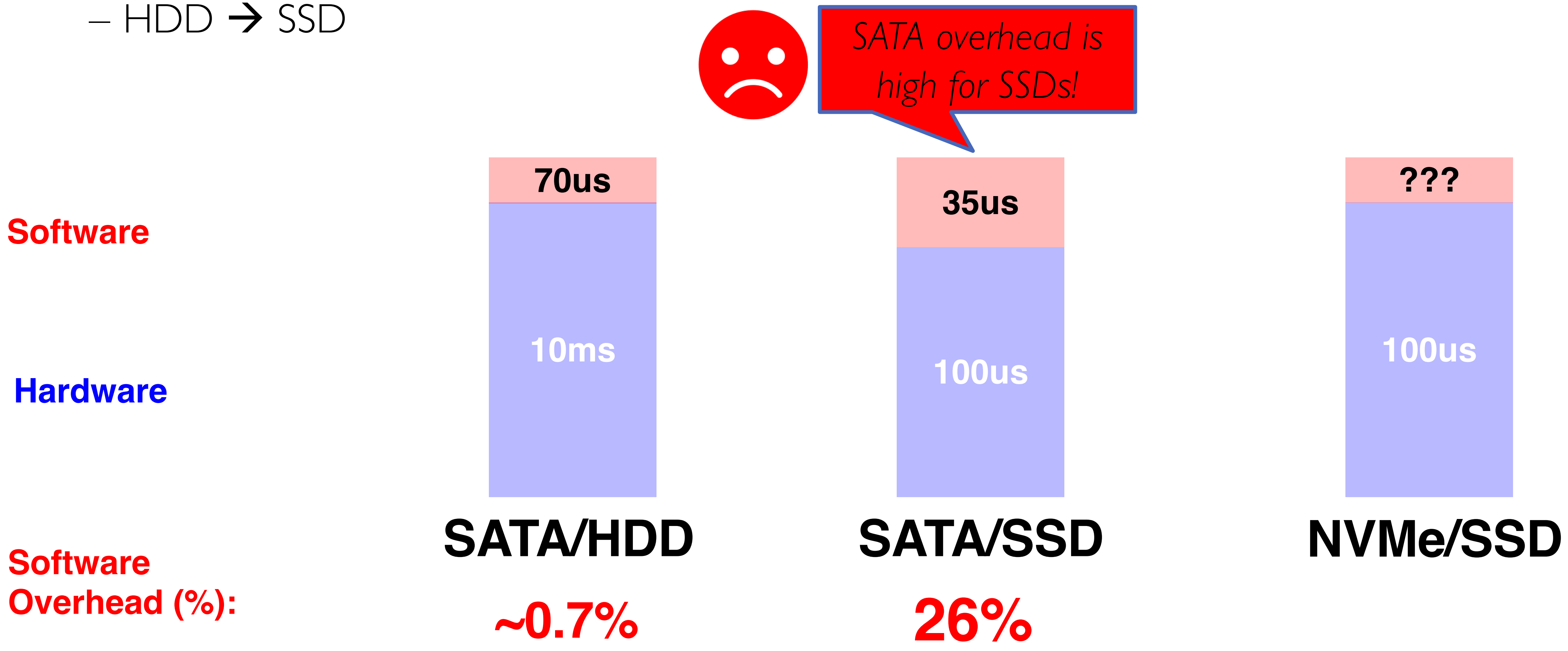
- Performance: reduce the legacy storage stack overhead
  - SATA
  - HDD → SSD



Software Overhead (%): **~0.7%** (SATA/HDD) **26%** (SATA/SSD)

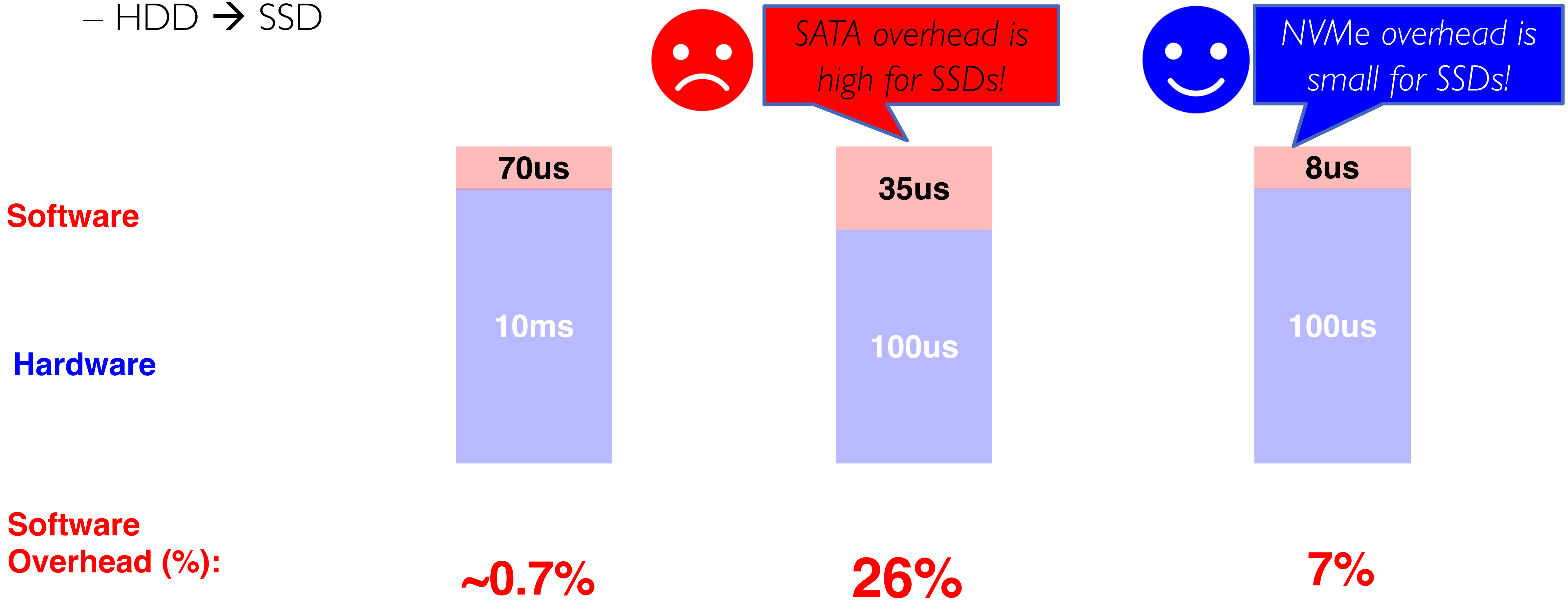
Why NVMe?

- Performance: reduce the legacy storage stack overhead
  - SATA → NVMe
  - HDD → SSD



Why NVMe?

- Performance: reduce the legacy storage stack overhead
  - SATA → NVMe
  - HDD → SSD



NVMe vs. SATA

Samsung SM951 SSDs

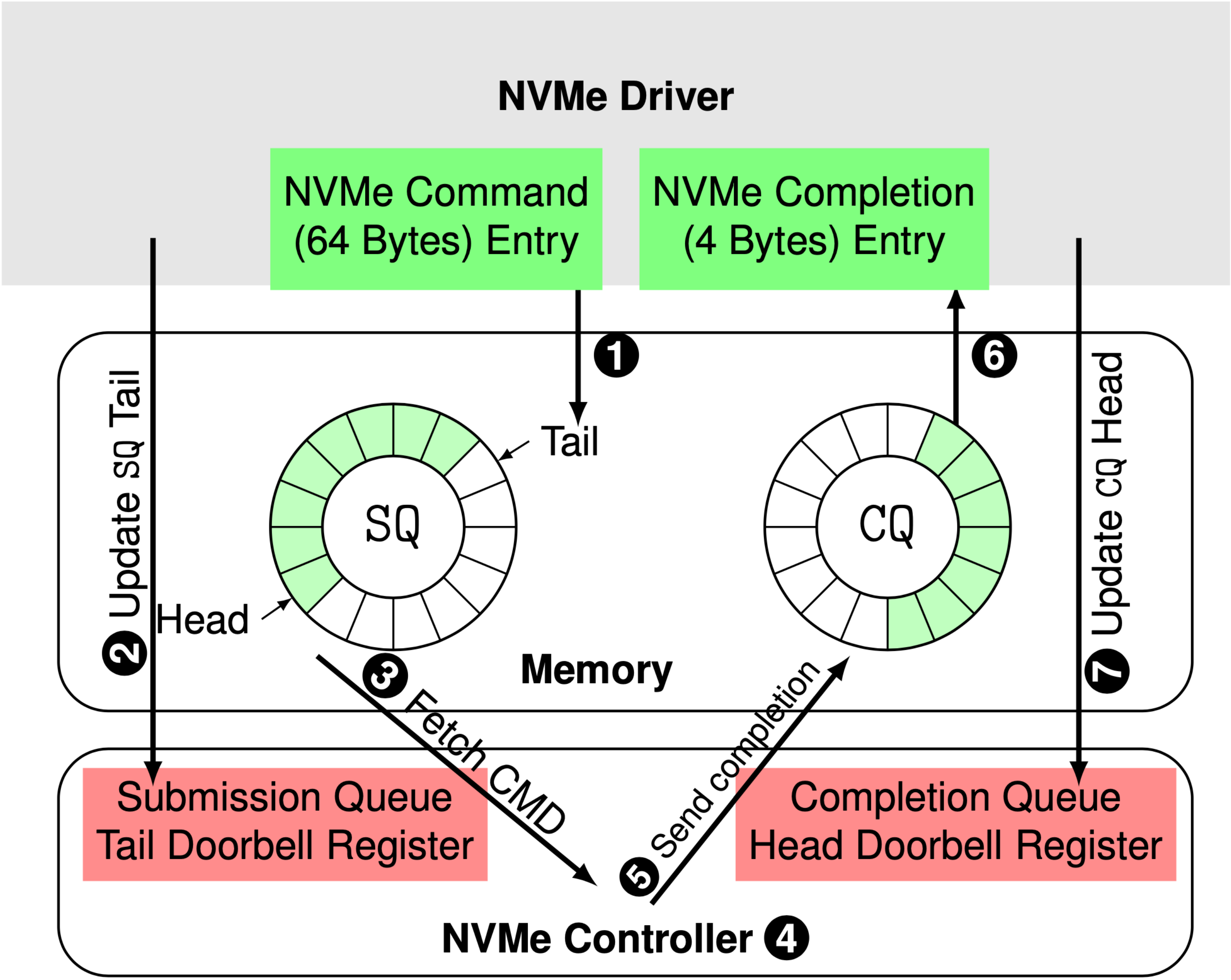
Measure performance in IOPS: I/Os per second

Under NVMe interface: 120K IOPS

Under SATA interface: 90K IOPS

33%

NVMe Processing Flow



Further Readings

- LWN: A block layer introduction part 1: the bio layer
- LWN: A block layer introduction part 2: the request layer
- Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems, SYSTOR13
- LWN: The multiqueue block layer
- LWN: Two new block I/O schedulers for 4.12
- LWN: The future of DAX
- Kernel Recipes 2017 - What's new in the world of storage for linux -Jens Axboe

RAID (Optional)

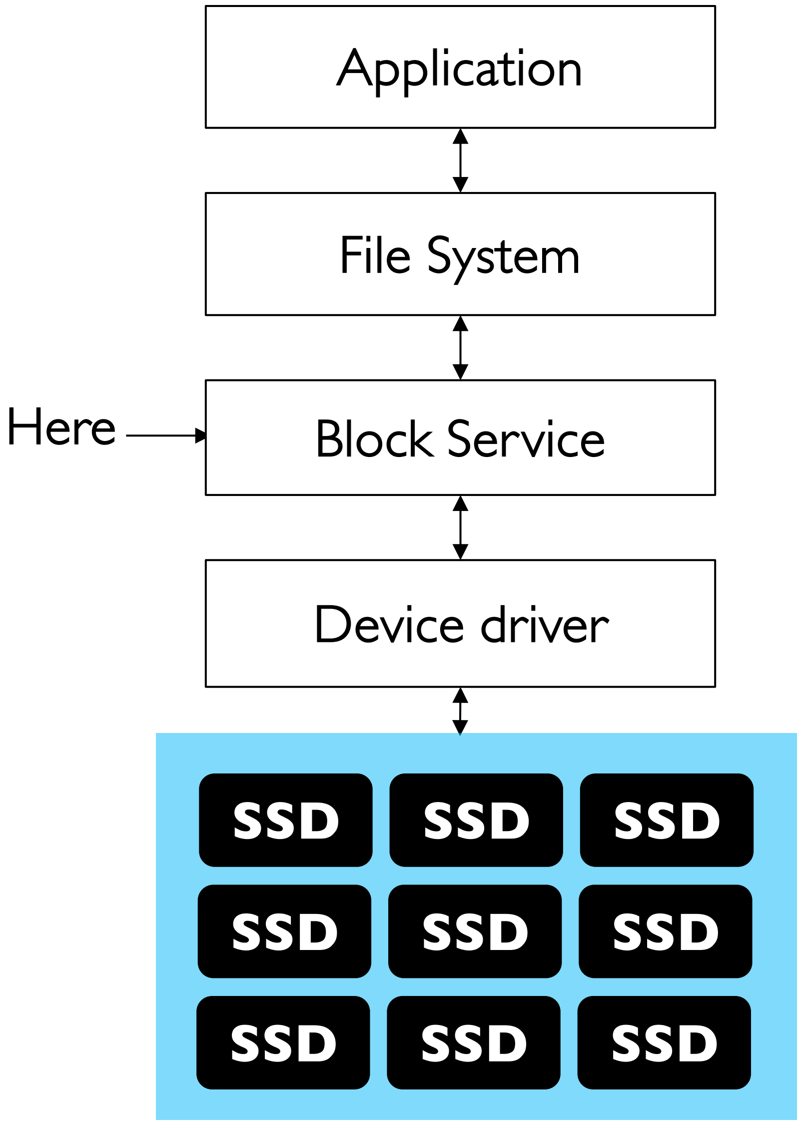
Wish Lists for Disks

- "Disk" can be SSD, HDD, etc.
  - Refer to block storage device (in contrast to byte-addressable devices, e.g., DRAM)
- Performance:
  - Faster
- Capacity:
  - Larger
- Reliability:
  - More reliable (or ideally a disk that never fails, "even when you shout at it" ☺)

Multi-disk Systems

- Reason 1: Storage capacity
  - Problem: cost, data growth
  - Solution: use multiple disks
- Reason 2: Performance
  - Problem: load balancing
  - Solutions: dynamic placement, striping
- Reason 3: Reliability
  - Problem: guaranteeing fault tolerance
  - Solutions: replication, parity

- Popular solution: RAID!

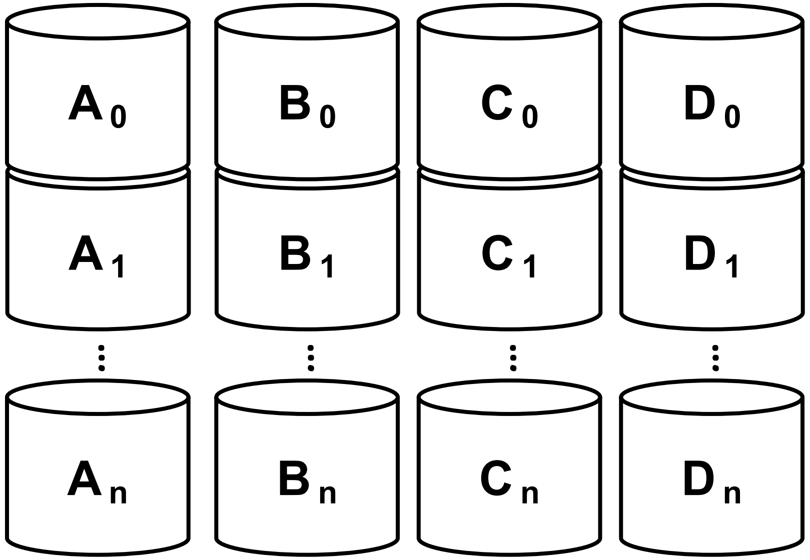


Example storage stack

Exposing Disk Arrays

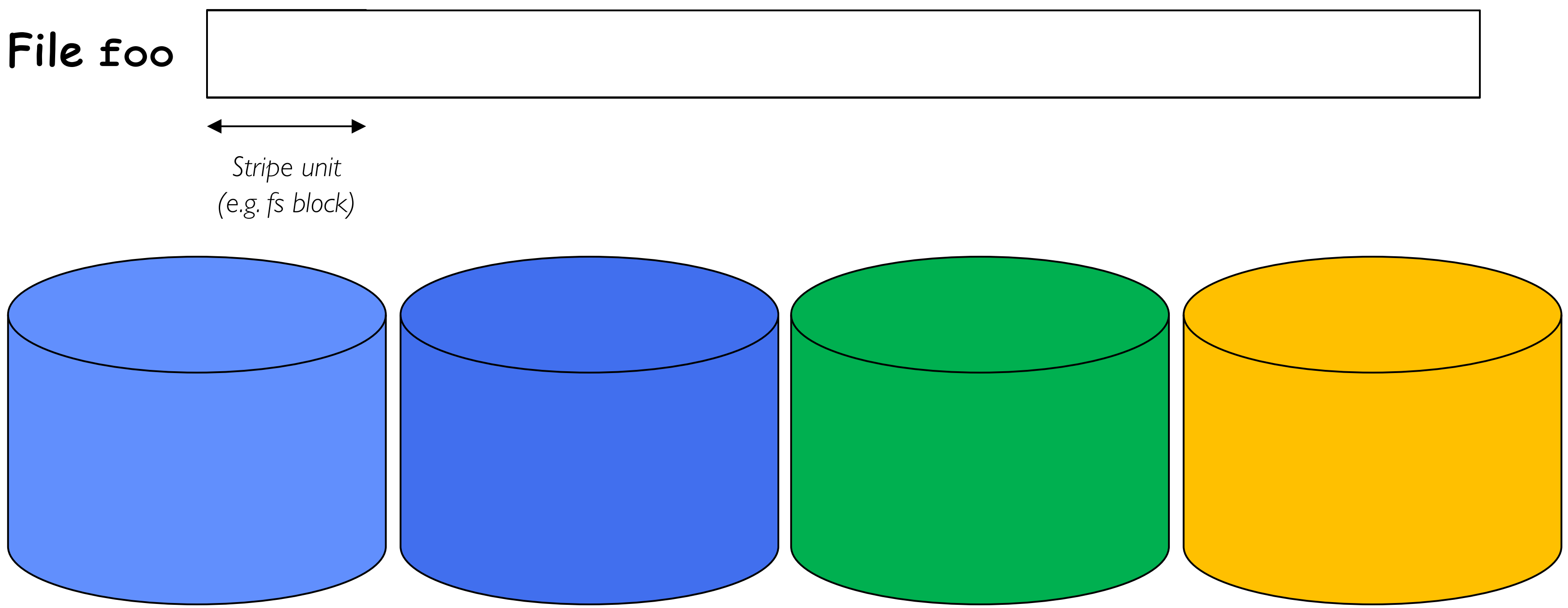
- Simplest solution: Just-a-Bunch-Of-Disks (JBOD)
  - Individual disks are exposed through controller
  - This is what you actually buy

JBOF -- for SSDs, "F" for "Flash"



RAID 0: Disk Striping

- Data interleaved across multiple disks
  - Large file streaming benefits from parallel transfers
    - "Large" defined relative to stripe unit
  - Thorough load balancing ideal for high-throughput requests
    - Hot file blocks get spread uniformly across all disks (good enough?)



Disk Striping 101

- How disk striping works
  - Break up LBN space into fixed-size stripe units
  - Distribute stripe units among disks in round-robin fashion
  - Straight-forward to compute location of block #B
    - Disk # = B % N, where % = modulo, N = number of disks
    - Disk block # = B / N (computes block offset on given disk)
- Key design decision: picking the stripe unit size
  - Assist alignment: choose multiple of file system block size

Too small

- Even small transfers span stripe units
- Extra seeks/accesses for little/no benefit (HDDs)

Too big

- No parallel transfers
- Poorer load balancing

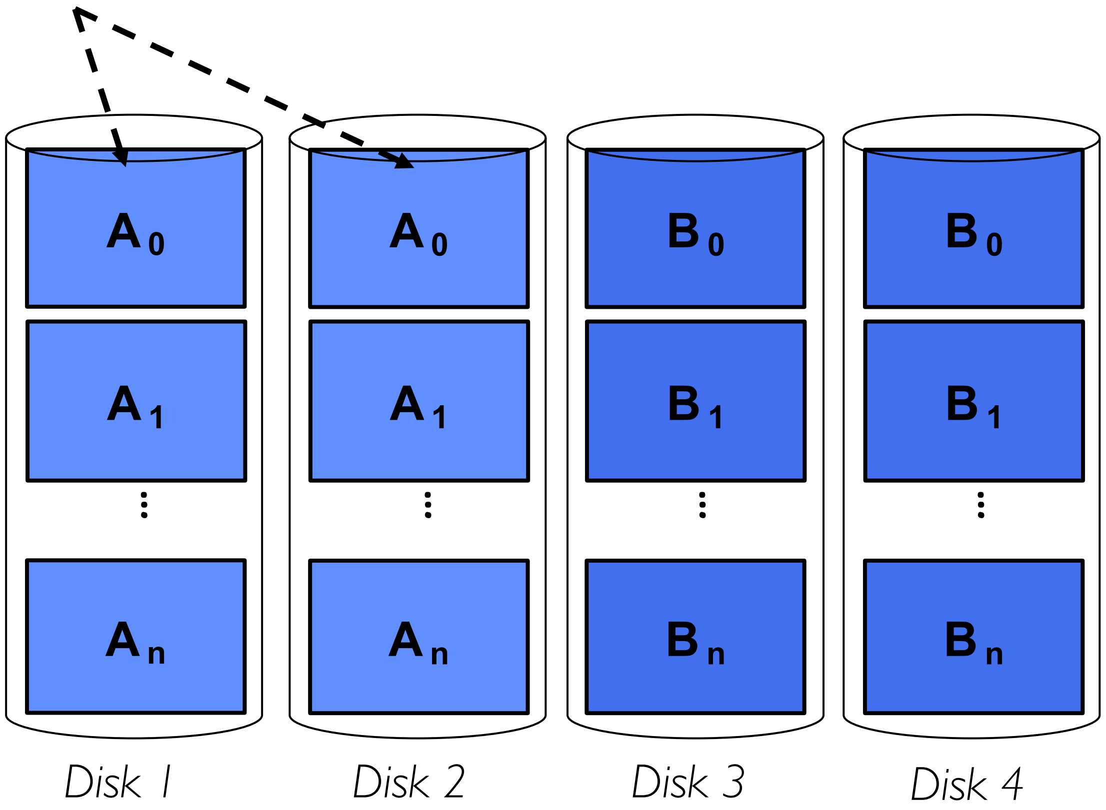
What Happens if a Disk Fails?

- In a JBOD (independent disk) system
  - All file systems on the disk are lost
- In a striped system
  - Part of each file system residing on failed disk is lost

- Backups can help, but are hard to get right
  - Backup scheduling is difficult
    - Choosing backup interval: how much data can you afford to lose?
  - Impact on performance while backing up
  - Storage provisioning for backup is non-trivial
    - Client data growth vs. number of backups stored

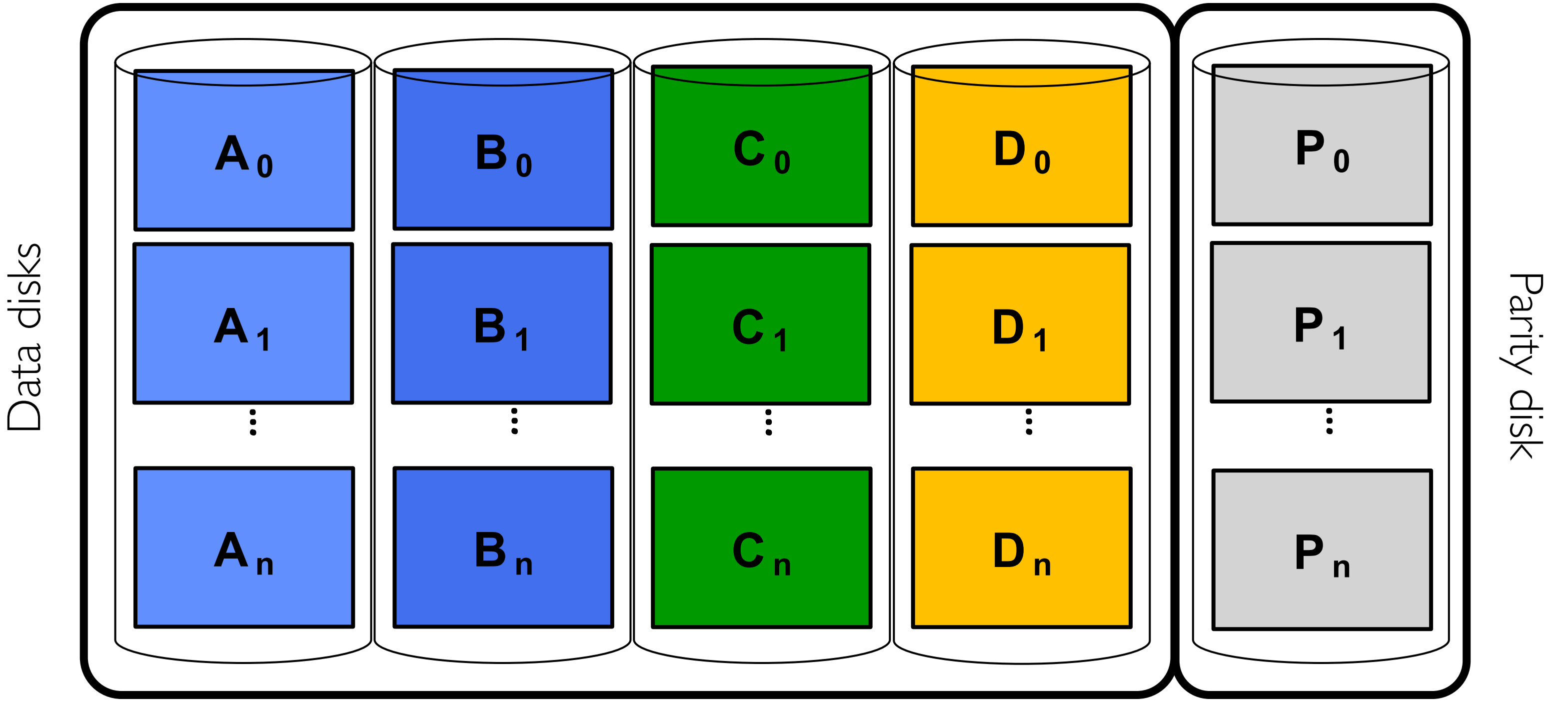
RAID-1: Mirroring (Redundancy via Replicas)

- Two (or more) copies of each write
  - Terms used: mirroring, shadowing, duplexing, etc.
- Write both replicas, read from either



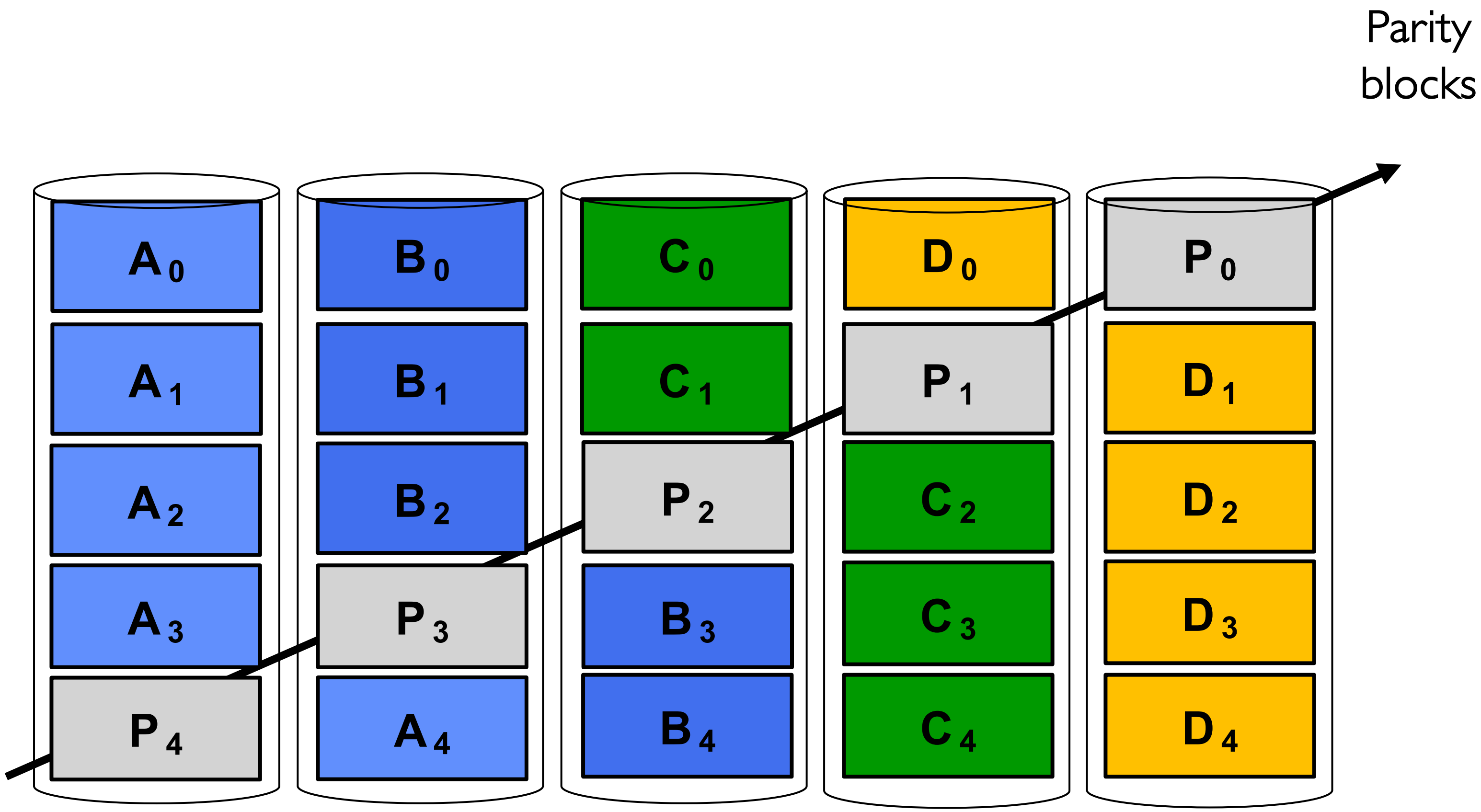
RAID-4: Parity Disks

- All writes update parity disk
  - downsides?



RAID-5: Striping the Parity

- Removes parity disk bottleneck
  - Parity is distributed across all disks



- Redundant Array of Inexpensive (or Independent) Disks
  - By UC-Berkeley researchers in late 80s (Garth Gibson)
- RAID 0 – Course-grained Striping with no redundancy
- RAID 1 – Mirroring of independent disks
- RAID 2 – Fine-grained data striping plus Hamming code disks
  - Uses Hamming codes to detect and correct multiple errors
  - Originally implemented when drives didn't always detect errors
  - Not used in real systems
- RAID 3 – Fine-grained data striping plus parity disk
- RAID 4 – Coarse-grained data striping plus parity disk
- RAID 5 – Coarse-grained data striping plus striped parity
- RAID 6 – Coarse-grained data striping plus 2 striped codes
- RAID N+3 – Coarse-grained data striping plus 3 striped codes
- Erasure Coding: more general …