I/O Approaches in Modern Storage Systems

CS647

John Malliotakis – jmal@ics.forth.gr

18 December, 2023



FORTH

INSTITUTE OF COMPUTER SCIENCE

Computer Architecture and VLSI Systems (CARV)

Outline

What we'll cover today:

- *Some* I/O internals
- Shifts in I/O framework design
- I/O approach taxonomy
- State of the art I/O methods and their semantics

Some terminology

- Storage devices $\rightarrow$ mainly block storage devices
  - E.g., HDDs, SSDs
  - Block: access unit
  - Blocks $\Longleftrightarrow$ sectors (HDD), pages (SSD)
- Last decade: also byte-addressable storage (NVM)

The illusion of fine granularity

```
paul@sith5 ~ % getconf PAGESIZE
4096
paul@sith5 ~ % fdisk -l /dev/nvme0n1

Disk /dev/nvme0n1: 1920.4 GB, 1920383410176 bytes, 3750748848 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

paul@sith5 ~ % fdisk -l /dev/sda
WARNING: fdisk GPT support is currently new, and therefore in an experimental phase. Use at your own discretion.

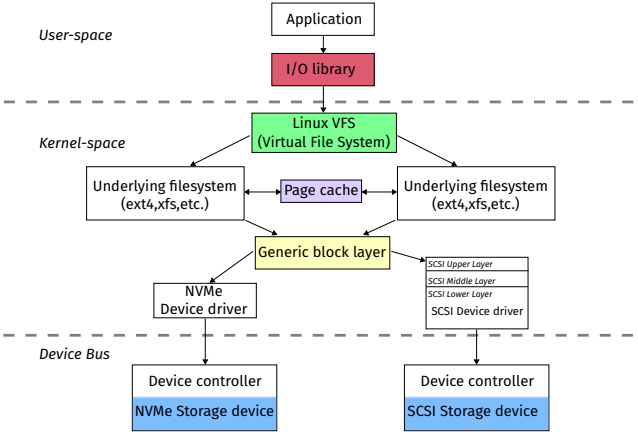
Disk /dev/sda: 120.0 GB, 120034123776 bytes, 234441648 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: gpt
Disk identifier: 6F401CD5-47AD-4BEB-AB16-D7C50D6A67D0


#         Start          End    Size  Type            Name
 1          2048      1050623    512M  EFI System
 2       1050624    234441613  111.3G  Linux filesyste
paul@sith5 ~ %
```

No such thing as fine-grain I/O!

- Each I/O software stack layer adds *abstraction*
- Practical reasons as well (*I/O efficiency*)

I/O software stack breakdown



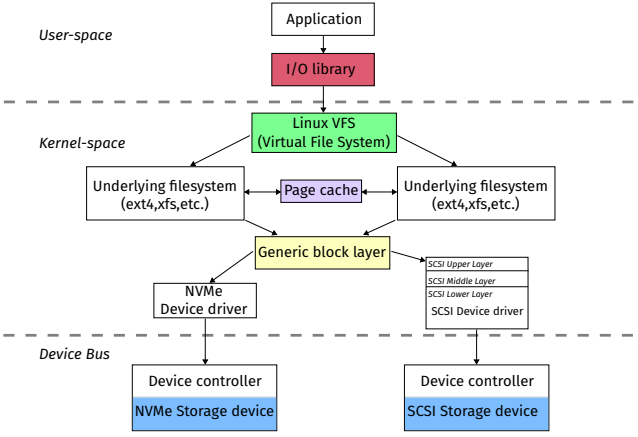
Call path through stack:

- I/O library $\overset{\text{system calls}}{\Longrightarrow}$ VFS
- VFS $\overset{\text{file operation handlers}}{\Longrightarrow}$ Underlying FS
- Underlying FS $\overset{\text{BIO}}{\Longrightarrow}$ Block layer

I/O units through stack:

- Application (I/O library): Byte arrays
- VFS/Underlying FS: Pages
- Block layer: Logical blocks
- Device driver: “Physical” blocks
- Device: Sectors/(SSD) pages

I/O software stack breakdown



The gist: per-layer *translations*

1. File offsets → OS pages
2. OS Pages → logical file blocks
3. (Drivers) Logical blocks → "physical" blocks
4. (Device controller) "Physical" blocks → sectors (HDD)/blocks/pages (SSD)

Classifying I/O approaches

Points of interest when examining I/O methods:

- User-space or kernel-space I/O?
- Where is data cached (if at all)?
- Write behaviour: when does data reach storage?
- Synchronous or asynchronous I/O?
  - (Asynchronous I/O) Polling or interrupts?

On kernel-space and user-space I/O

- Historically: **kernel-space I/O**, *syscall-based* interface
- Kernel offers security, isolation (via FS interface)
- Trap (per syscall) + software stack cost $\ll$ *Device latency*
- *Newer, faster* devices $\rightarrow$ software stack bottleneck $\rightarrow$ user-space approaches

On data caching (and copying)

Traditionally:

- DRAM caching $\rightarrow$ hide device latency
- OS caching: kernel-space page cache $\rightarrow$ 2 data copies (*user-space* buffer, *kernel-space* cache)

On data caching (and copying)

State of the art: self-caching applications

- End-to-end approach: finer-grained, application-based caching
- Also allows explicit writeback control
  - Difficult with traditional OS-based techniques
  - No write-ordering provided by the kernel
- Additionally: excessive data copying wastes CPU cycles
  - Especially if copying from remote memory on NUMA systems
- Thus: zero-copy I/O techniques

With NVM: caching often(?) *unwanted*

- Latency *similar* (but larger) to DRAM

On write behaviour

With OS DRAM caching:

- *Writeback*: write propagation, DRAM → storage
- Understanding the semantics: important for recoverability guarantees!
- Writeback is not immediate on write syscall!

With user-space approaches/self caching:

- Writes: directly propagated to storage

Directly propagated writes → predictable behaviour/latency

On synchronous and asynchronous I/O

With synchronous (blocking) I/O:

- Application thread stalls on I/O request issue
- Resumes execution once I/O is finished
- \+ *Familiar* (imperative) programming model, simplicity
- \- Can underutilize CPU even if work is available

With asynchronous I/O:

- Control returned immediately to application thread on I/O request issue
- The goal: I/O overlap with computation
- \+ Maximize potential CPU and device **utilization**
- \- Increased complexity (I/O ordering, latency guarantees, among others)

Synchronous I/O can be used to emulate asynchronous I/O (e.g., POSIX AIO)

Polling vs. interrupts

Polling Manually check I/O status

- Check possible from user-space or kernel-space
- Periodic polling: *wasted* CPU cycles
- Instead: explicit (on-demand) polling

Interrupts Device sends CPU interrupt on I/O completion

- Minimal CPU involvement (on completion only)
- Interrupt $\rightarrow$ interrupt handling (i.e., trap) cost

I/O method overview

Terminology:

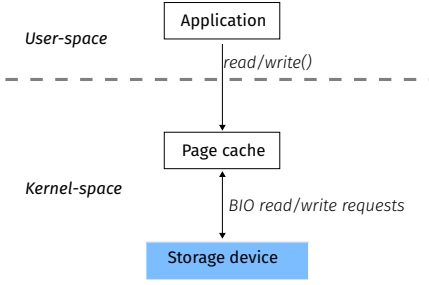
BIO	Buffered I/O
DIO	Direct I/O
MMIO	Memory-mapped I/O
URING	I/O through *io_uring* interface
SPDK	Storage Plane Development Kit, Intel NVMe library
DAX	Direct Access, designed for byte-addressable devices

I/O method overview

Method classification:

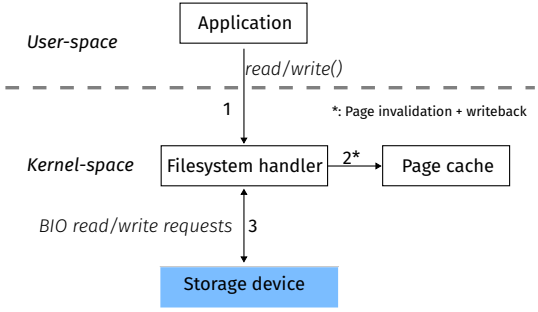
	Kernel/User-space	Data copies	Sync/Async
BIO	K	Page cache + User buffer	S
DIO	K	User buffer only	S
MMIO	K	Page cache only	S
URING	K	BIO/DIO	A
SPDK	U	User buffer only	A
DAX	U	Device only	S

Buffered I/O



- Application issues traditional *read*/*write* syscall
  - Variants: *pread*/*pwrite*: custom offset, *preadv*/*pwritev*: vectored I/O (scatter gather)
- Kernel services system call synchronously via VFS, underlying FS
- Read path: storage device → kernel page cache $\overset{\text{data copy}}{\Longrightarrow}$ user buffer
- Write path: user buffer $\overset{\text{data copy}}{\Longrightarrow}$ kernel page cache
- System call per transfer

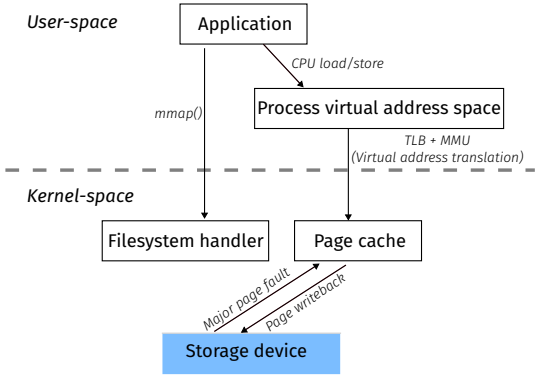
Direct I/O



Direct I/O:

- Files opened with `O_DIRECT` flag
- Same API as BIO
- Goal: bypass page cache
- Intended user: self-caching applications
- Data moved between user-space buffer and storage
- Requires user buffer alignment, specific transfer sizes
- Page cache interaction semantics!

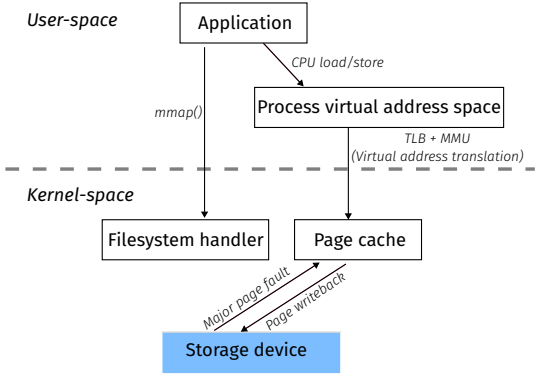
Memory-mapped I/O



Memory-mapped I/O

- Map file/device ranges to process address space → *file-backed* mappings
- Virtual address space → transparent data representation
- Fast path: zero software overhead (address translation via TLB + MMU)
- Slow path: TLB miss + page fault
  - Page cache hit → minor fault
  - Page cache miss → major fault
  - Major faults require synchronous page I/O + PTE setup

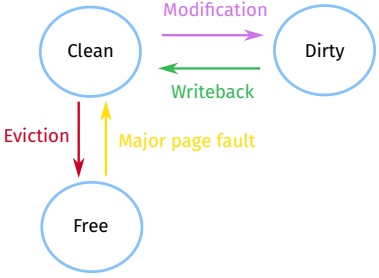
Memory-mapped I/O



On anonymous mappings:

- Anonymous $\iff$ backed only by DRAM, no file
- Same mechanisms as file-backed mappings
- Essentially virtual address range reservations
- Physical memory allocated on first touch via page fault
- I/O involved only when swapping
- All *malloc* implementations use anonymous `mmap`

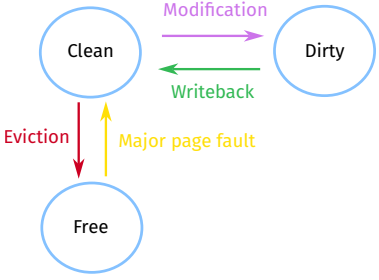
Kernel page cache semantics



Possible page states and their transitions

- Pages allocated on cache misses
- Initially *clean*, marked *dirty* on modification
- The cache evicts (reclaims) only clean pages (no I/O cost)
- File-backed mappings → only regular pages on Linux → we address this with *xmap*

Kernel page cache semantics



Possible page states and their transitions

Writebacks to storage are not guaranteed

- Explicitly requested via *sync* syscall family
- Writebacks triggered by kernel on memory pressure (need clean pages for reclaim)
- On great memory pressure: application stalls with synchronous writebacks
- Kernel also periodically performs writeback (dirty inode tracking)
- Writeback behaviour tunable via *sysfs*

Application interaction with the Linux (POSIX) page cache

Readahead: adaptive page prefetching

- Synchronous or asynchronous
- Triggered by kernel on BIO read/write, MMIO page fault
- Based on previously accessed pages
- Sliding window technique (similar to TCP)

Application interaction with the Linux (POSIX) page cache

The page cache accepts usage hints

- Via *fadvise* (BIO), *madvise* (MMIO)
- Notable hints:

SEQUENTIAL	Enable readahead
RANDOM	Disable readahead
WILLNEED	Explicit readahead
DONTNEED	Mark candidate pages for reclaim

io_uring

io_uring:

- Linux kernel asynchronous I/O interface
- Replaced older, less performant AIO mechanism
- Application submits I/O requests via syscall (*io_uring_enter*)
- Kernel performs underlying I/O, either BIO or DIO
- **Three** operating modes:
  1. *Interrupt driven*: I/O completions reaped with no syscall
  2. *Polled*: Completion polling on io_uring_enter – DIO only
  3. *Kernel polled*: kernel thread polls for submissions → no syscall submissions
     - Kernel thread sleeps when idle → woken with io_uring_enter

io_uring

Implementation notes:

- Two queues (implemented as ring buffers)
- Submission queue (SQ), Completion queue (CQ)
- Queues: single producer, single consumer $\overset{\text{multiple queue-pairs}}{\Longrightarrow}$ Multithreading
- Application memory maps queues $\rightarrow$ Queue memory shared with kernel $\rightarrow$ no metadata copying

lib_uring: user-space io_uring wrapper

SPDK

- Intel Storage Plane Development Kit
- Targets NVMe devices
- User-space I/O library → **no** kernel involvement
- Requires user-space ownership over device

Implementation notes:

- NVMe device unbound from kernel driver via *sysfs*, then bound to uio/vfio device driver
  - *uio*: allows user-space access to device memory (e.g., control registers, device RAM)
  - *vfio*: newer, adds IOMMU (virtualized I/O addresses) support
- Single thread queue-pair approach, similar to io_uring
- I/O requests submitted via API call along with completion callback
- Completions explicitly polled via API call
- DMA used for transfers → I/O buffers must be *DMAable*, allocated via *spdk_dma_malloc*

DAX

Direct Access:

- Designed for byte-addressable storage devices, e.g., NVM
- Bypasses kernel page cache
- Requires DAX-capable filesystem (ext4, xfs)

Use case: `mmap` + DAX filesystem

- **Map** process virtual address range directly to device range
- Direct mapping → address translations known at map time
- Page faults to setup (huge) page table entries for these translations
- Or MAP_POPULATE flag on `mmap` call

Resources – Further reading

Most info on these slides came from:

- BIO, DIO, MMIO: *Understanding the Linux Kernel, 3rd Edition*
  - Chapters 9, 12, 14, 15, 16
- io_uring: `https://kernel.dk/io_uring.pdf`
- SPDK: `https://spdk.io/doc`
- DAX: `https://kernel.org/doc/html/latest/filesystems/dax.html`