LINUX PLUMBERS CONFERENCE > September 20-24, 2021

Building a fast NVMe passthru

Kanchan Joshi

Memory, Samsung Semiconductor (SSIR)

Outline

- NVMe Generic Device: why and what
- Async IOCTL: user-interface and under-the-hood
- NVMe: Moving from sync passthru ~~to~~ async uring passthru
- Feedback / Opens / Next steps

Credits

- io-uring: for being around
- Maintainers (Jens, Christoph, Keith) & the mailing-list: for all the directions & feedback so far

NVMe block-interface

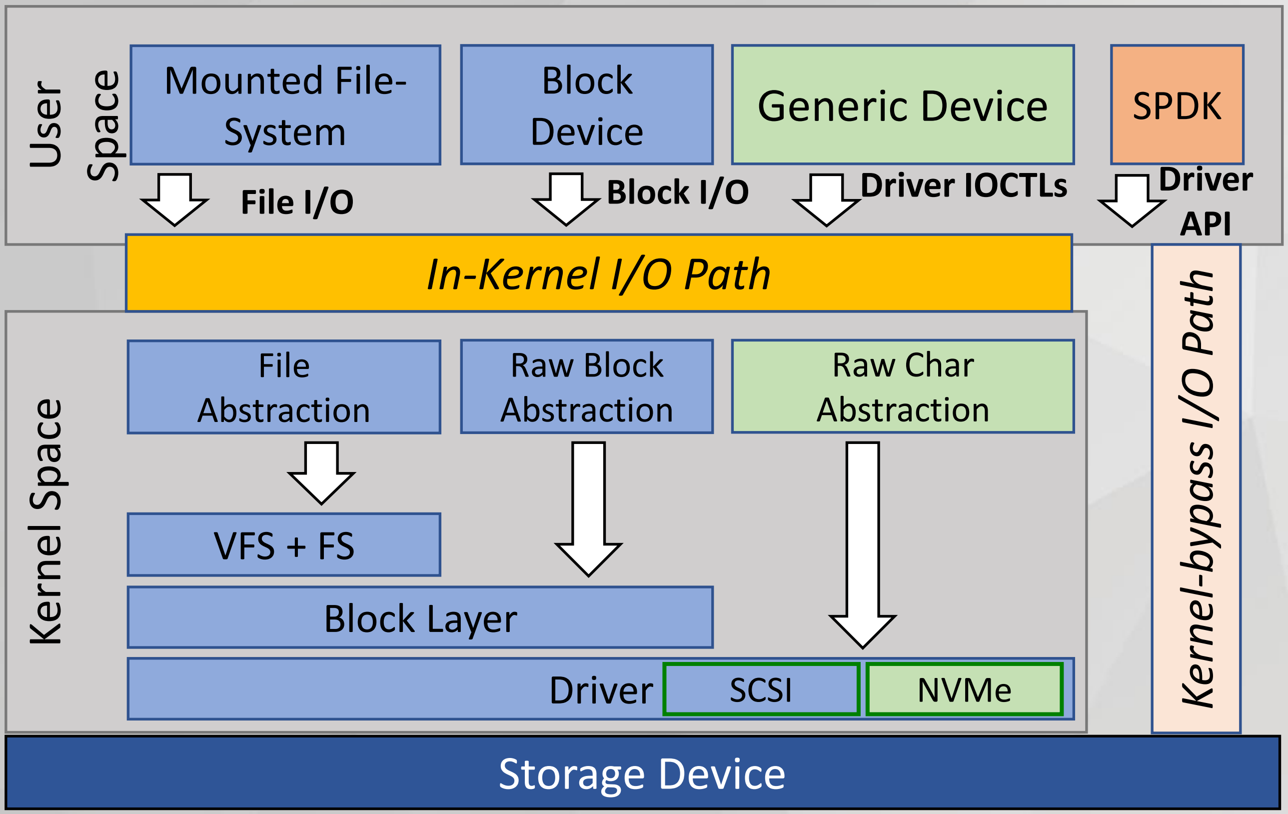
- Subject to conditions/rules
  - Block-device with zero capacity
  - Block-device marked as read-only
  - Block-device marked hidden
- This generally happens when
  - Device contains a feature that kernel does not support (e.g. unsupported format/PI)
  - New device/command-set types (e.g. KV, ZNS)

```
nvme-cli $blockdev --getsz /dev/nvme0n1
0
nvme-cli $./nvme list -v
NVM Express Subsystems

Subsystem        Subsystem-NQN
---------------- ------------------------------------------------------------------
nvme-subsys0     nqn.2019-08.org.qemu:deadbeef

NVM Express Controllers

Device   SN                   MN                                       FR       T
-------- -------------------- ---------------------------------------- -------- -
nvme0    deadbeef             QEMU NVMe Ctrl                           1.0      p

NVM Express Namespaces

Device       Generic      NSID     Usage                      Format           Co
------------ ------------ -------- -------------------------- ---------------- --
nvme0n1      ng0n1        1        12.88  GB /  12.86  GB     4 KiB +  8 B
nvme0n2      ng0n2        2        12.88  GB /  12.88  GB     4 KiB +  0 B
```

New kid on the ~~block~~ char!

NVMe generic device

NVMe Generic Interface



- Per-namespace char device (/dev/ngXnY)
- Upstream in NVMe (5.13)
- Always available
- In-kernel path (unlike SPDK) for early adopters of technology/features

Using the NVMe char device

- Nvme-cli can enumerate and do all that it can do on block-device
- Usable over NVMeOF
  - Automatic, when block interface (/dev/nvme0n1) is up
  - When not, available after enabling controller passthru (CONFIG_NVME_TARGET_PASSTHRU)

```
# Set device nvme0 as the controller we want to expose over the fabric
echo -n /dev/nvme0 > /sys/kernel/config/nvmet/subsystems/testnqn/passthru/device_path
echo 1 > /sys/kernel/config/nvmet/subsystems/testnqn/passthru/enable
```

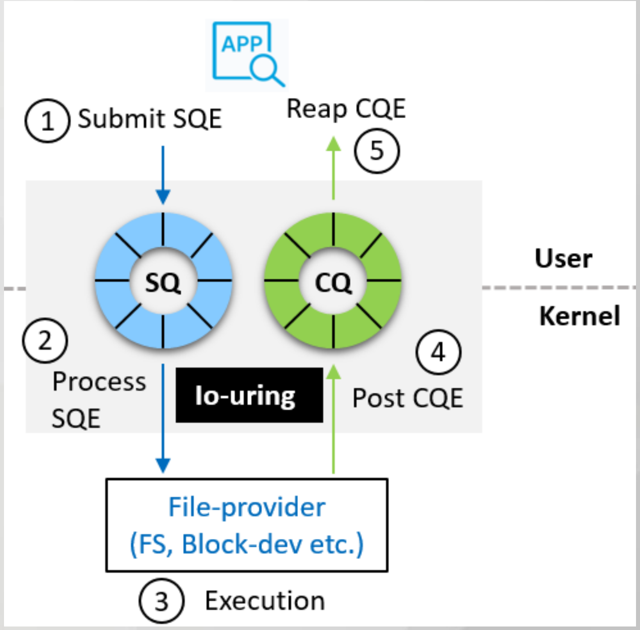
- Application can send any NVMe command via passthru interface
  - Current transport: via IOCTL, which isn't great!
  - Future transport: io_uring

```
static const struct file_operations nvme_ns_chr_fops = {
        .owner          = THIS_MODULE,
        .open           = nvme_ns_chr_open,
        .release        = nvme_ns_chr_release,
        .unlocked_ioctl = nvme_ns_chr_ioctl,
        .compat_ioctl   = compat_ptr_ioctl,
};
```

Turns out Jens had already set about turning ioctl async; in io-uring way

io_uring: in a nutshell

- Scalable asynchronous IO infrastructure
  - File IO as well as Network IO
  - Async without needing O_DIRECT
  - Extensible - rapidly adding async variants of sync syscalls
  - mkdir, link, symlink: few recent ones
- User/Kernel interface
  - Communication backbone: shared ring-buffers (SQ and CQ)
  - Reduce syscalls & copies
  - Programming model
    - Prepare IO: Get SQE from SQ ring, and fill it up (fill more to make a batch)
    - Submit IO: By calling io_uring_enter
    - Complete IO: Reap CQE from CQ ring
  - Submission can be offloaded (no syscall)
  - Completion can be polled (interrupt-free IO)

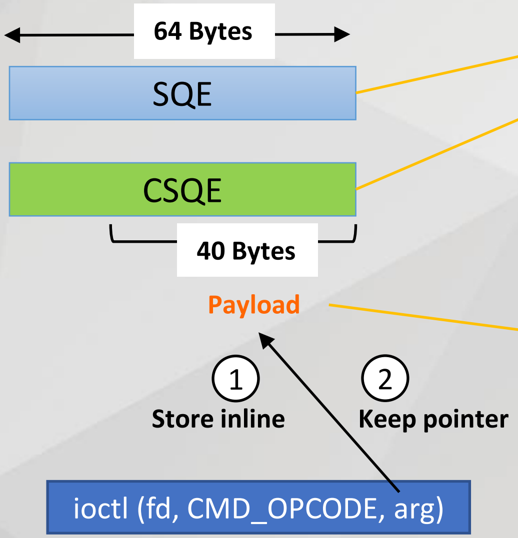


Faster IO through io_uring: https://kernel-recipes.org/en/2019/talks/faster-io-through-io_uring

LINUX PLUMBERS CONFERENCE › September 20-24, 2021

Async ioctl: user-interface

- Uring-cmd: IOCTL-like facility
- New opcode IORING_OP_URING_CMD
- New ‘command’ SQE (CSQE) to be used
  - CSQE = Specialized SQE with 40 bytes of free-space. Useful for avoiding allocation (for IOCTL cmd) cost
  - Can be used in other way too (e.g. pointer to larger IOCTL cmd)
  - io_uring passes the payload to ioctl provider




```c
struct uring_cmd_ioc {
        __u32   ioctl_cmd;
        __u32   unused1;
        __u64   unused2[4];
};

static int get_bs(struct io_uring *ring, const char *dev)
{
        struct io_uring_cqe *cqe;
        struct io_uring_sqe *sqe;
        struct io_uring_cmd_sqe *csqe;
        struct uring_cmd_ioc *ucmd;
        int ret, fd;

        fd = open(dev, O_RDONLY);

        sqe = io_uring_get_sqe(ring);
        csqe = (void *) sqe;
        memset(csqe, 0, sizeof(*csqe));
        csqe->hdr.opcode = IORING_OP_URING_CMD;
        csqe->hdr.fd = fd;
        csqe->user_data = 0x1234;
        csqe->op = BLOCK_URING_OP_IOCTL;
        ucmd = (void *) &csqe->pdu;
        ucmd->ioctl_cmd = BLKBSZGET;

        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);
        printf("bs=%d\n", cqe->res);
        io_uring_cqe_seen(ring, cqe);
        return 0;
}
```

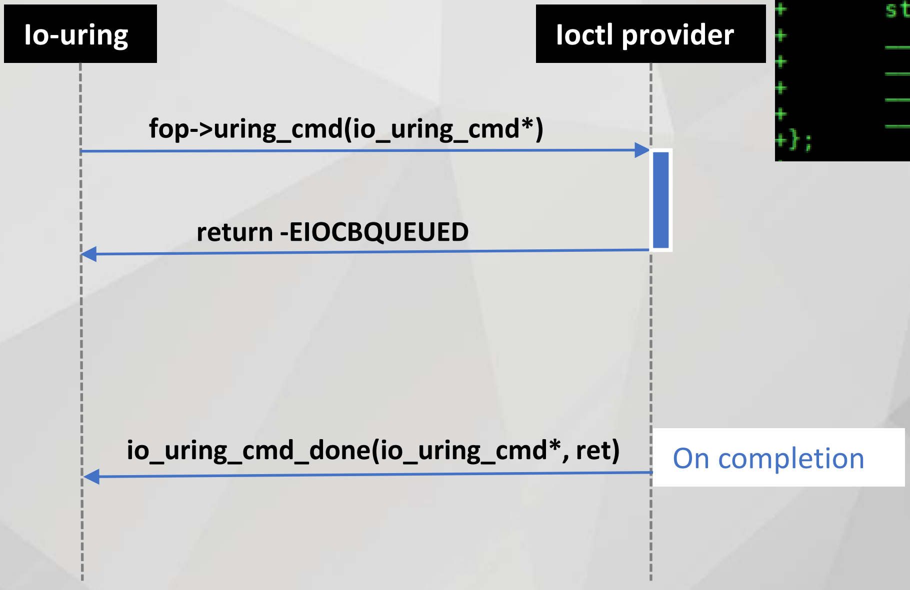
Jens v4 series: https://lore.kernel.org/linux-nvme/20210317221027.366780-1-axboe@kernel.dk/

Async ioctl: inside kernel

- Ioctl provider is expected to implement new *uring_cmd* method in file_operations
- Io_uring fetches CSQE, and prepares ‘struct io_uring_cmd’ out of it; this is used for all further communication
  - Submit ioctl by fop->uring_cmd
  - Provider does what it should, and returns without blocking
  - It can return result instantly, or defer
  - For the latter, it returns by calling io_uring_cmd_done()
  - Io_uring collects the result, and post that into CQE

```
struct file_operations {
        struct module *owner;
@@ -2059,6 +2068,8 @@ struct file_operations {
                        struct file *file_out, loff_t pos_out,
                        loff_t len, unsigned int remap_flags);
        int (*fadvise)(struct file *, loff_t, loff_t, int);
+
+       int (*uring_cmd)(struct io_uring_cmd *, enum io_uring_cmd_flags);
} __randomize_layout;
```

```
+struct io_uring_cmd {
+       struct file     *file;
+       __u16           op;
+       __u16           unused;
+       __u32           len;
+       __u64           pdu[5];
+};
```

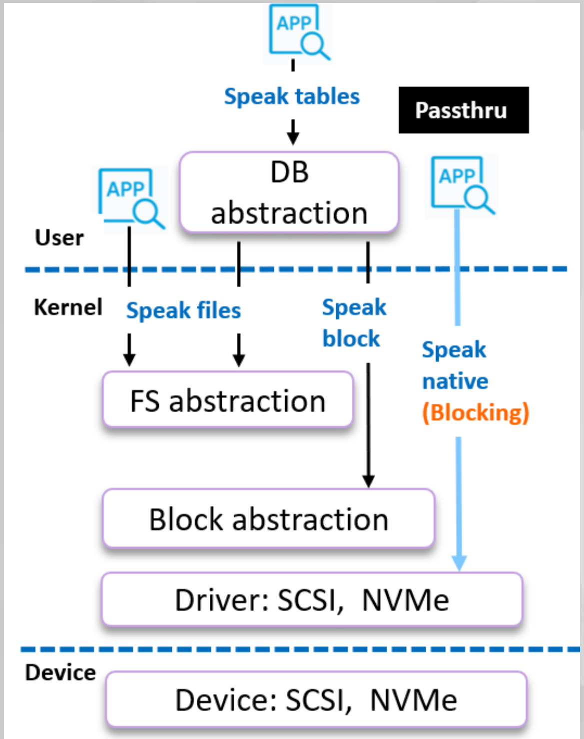


Async ioctl: use cases

- Network IO
- Storage:
  - FS users, ioctl-heavy applications e.g. xfs-scrub
  - Passthru – already a lean path to storage; make it useful → **Rest of the slides cover this!**
  - Other suggestions?

IOCTL passthru → Uring passthru

NVMe passthru: Good and Bad



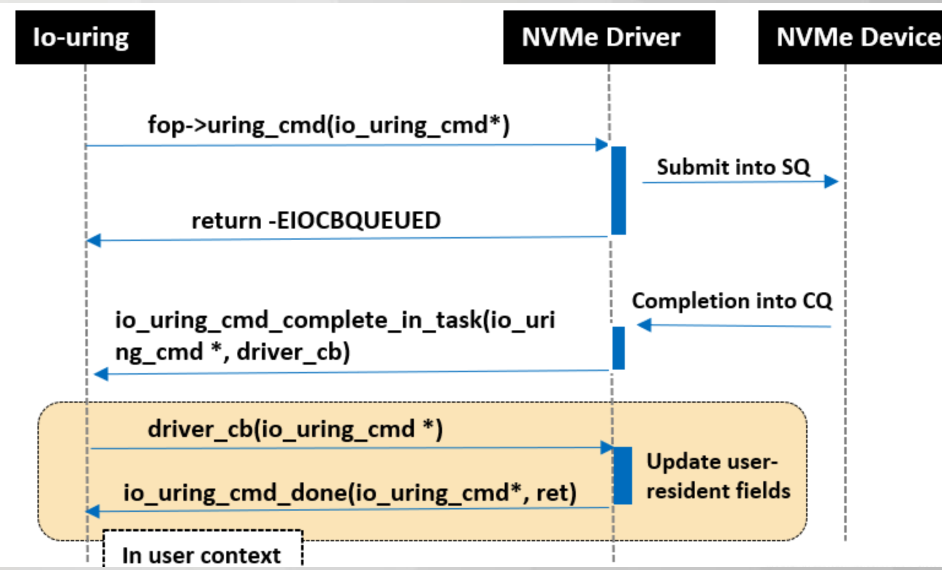
- New features in NVMe are emerging fast
- Stacked layers
  - we have few abstractions stacked upon the storage device; Each has its purpose & utility (great for general purpose)
  - May take some time/consensus-building for device-feature to move up the ladders of abstractions, and show up to user-space
  - At times, opaqueness need to be explicitly crafted (for future reuse) while building file/user interface over new device interfaces
  - This presents challenges for early technology adopters
- With passthru interface, Kernel provides a way to skip the layers
  - Allows new features to be consumed (in native way at least) without having to build block-generic commands, in-kernel users/emulations and user-interfaces
  - Potential path for building domain-specific application (app-specific FS/DB)
- But passthru travels via blocking ioctl – virtually useless for fast NVMe devices ☹

Nvme passthru: wire up async transport

- Current nvme ioctl operation
  - NVMe interface is 'naturally' async
  - Host submit command into NVMe SQ at time T; Device sends back completion separately in NVMe CQ at T+ Δ T
  - Driver implements sync-over-async by forcing submitter go into blocking-wait

- Uring-cmd based operation:
  - Driver decouples completion from submission; no blocking-wait
  - Async-update-to-user-memory problem
    - General problem if ioctl-cmd has some fields which need to be updated on I/O completion
    - Such fields cannot be touched if completion is arriving in interrupt-context!
  - Thankfully there is task-work infra in Kernel
    - Driver sets up a callback to do all the update; passes that to io_uring
    - Io_uring sets up a task-work, that executes driver-defined callback

```
static const struct file_operations nvme_ns_chr_fops = {
        .owner          = THIS_MODULE,
        .open           = nvme_ns_chr_open,
        .release        = nvme_ns_chr_release,
        .unlocked_ioctl = nvme_ns_chr_ioctl,
        .compat_ioctl   = compat_ptr_ioctl,
        .uring_cmd      = nvme_ns_chr_async_ioctl,
};
```



Example

- Read from /dev/ng0n1

Allocate and setup nvme passthru command

Prepare CSQE for uring-cmd

Setup passthrough ioctl & cmd pointer inside uring-cmd

- Tidbits for ZNS
  - Async zone-reset
  - Zone-append at multi-QD

```
/* this overlays struct io_uring_cmd pdu (40 bytes) */
struct nvme_uring_cmd {
        __u32   ioctl_cmd;
        __u32   unused1;
        void    *argp;
};
/* issue passthru command to read from device into buf */
void nvme_passthru_read(struct io_uring *ring, void *buf)
{
        struct io_uring_sqe *sqe = NULL;
        struct io_uring_cqe *cqe = NULL;
        struct io_uring_cmd_sqe *csqe;
        struct nvme_passthru_cmd *ptcmd;
        struct nvme_uring_cmd *ncmd;
        int fd;

        fd = open("/dev/ng0n1", O_RDONLY);

        ptcmd = (struct nvme_passthru_cmd *)malloc(sizeof(struct nvme_passthru_cmd));
        prepare_pt_cmd(ptcmd, buf);

        sqe = io_uring_get_sqe(ring);
        csqe = (void *)sqe;
        csqe->hdr.fd = fd;
        csqe->hdr.opcode = IORING_OP_URING_CMD;
        csqe->user_data = 0x1234;

        ncmd = (void *) &csqe->pdu;
        ncmd->ioctl_cmd = NVME_IOCTL_IO64_CMD;
        ncmd->argp = (void *)ptcmd;

        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);

        printf("res=%d\n", cqe->res);
        io_uring_cqe_seen(ring, cqe);
        free(ptcmd);
}
```

Features for faster IO

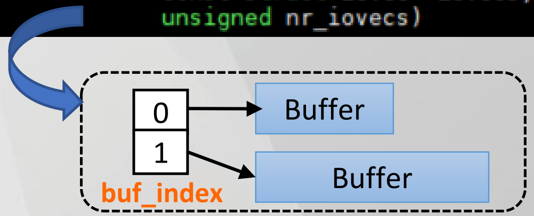
- Async is first step
- Since NVMe is talking to io_uring, there is room for more

Feature	What it does	Io_uring	Uring-passthru
Register-files	Reference fd once and reuse	☑	☑
SQPoll	Offload IO submission	☑	☑
Fixed-buffer	Map IO buffer once and reuse	☑	☒
Async polling	Interrupt-free completion	☑	☒

Uring passthru: fixed buffer

- How fixed-buffer helps
  - Pin once (*pin_user_pages*), reuse the buffer: reduce per-io cost for pin/unpin
  - Io_uring_register() to pin N buffers upfront; basically setup up bio_vec for these buffers
  - Specify IO (fixed-buffer opcode) by using any of the pre-mapped buffer

```
int io_uring_register_buffers(struct io_uring *ring,
                              const struct iovec *iovecs,
                              unsigned nr_iovecs)
```



- Io_uring plumbing
  - New opcode IORING_OP_URING_CMD_FIXED
  - Buffer are registered as before, and sqe->buf_index to be used for IO
  - Make the corresponding bio_vec accessible to driver

```
sqe = io_uring_get_sqe(ring);
csqe = (void *)sqe;
csqe->hdr.fd = fd;
csqe->hdr.opcode = IORING_OP_URING_CMD_FIXED;
csqe->buf_index = buf_index;
csqe->user_data = 0x1234;

ncmd = (void *) &csqe->pdu;
ncmd->ioctl_cmd = NVME_IOCTL_IO64_CMD;
ncmd->argp = (void *)ptcmd;
```

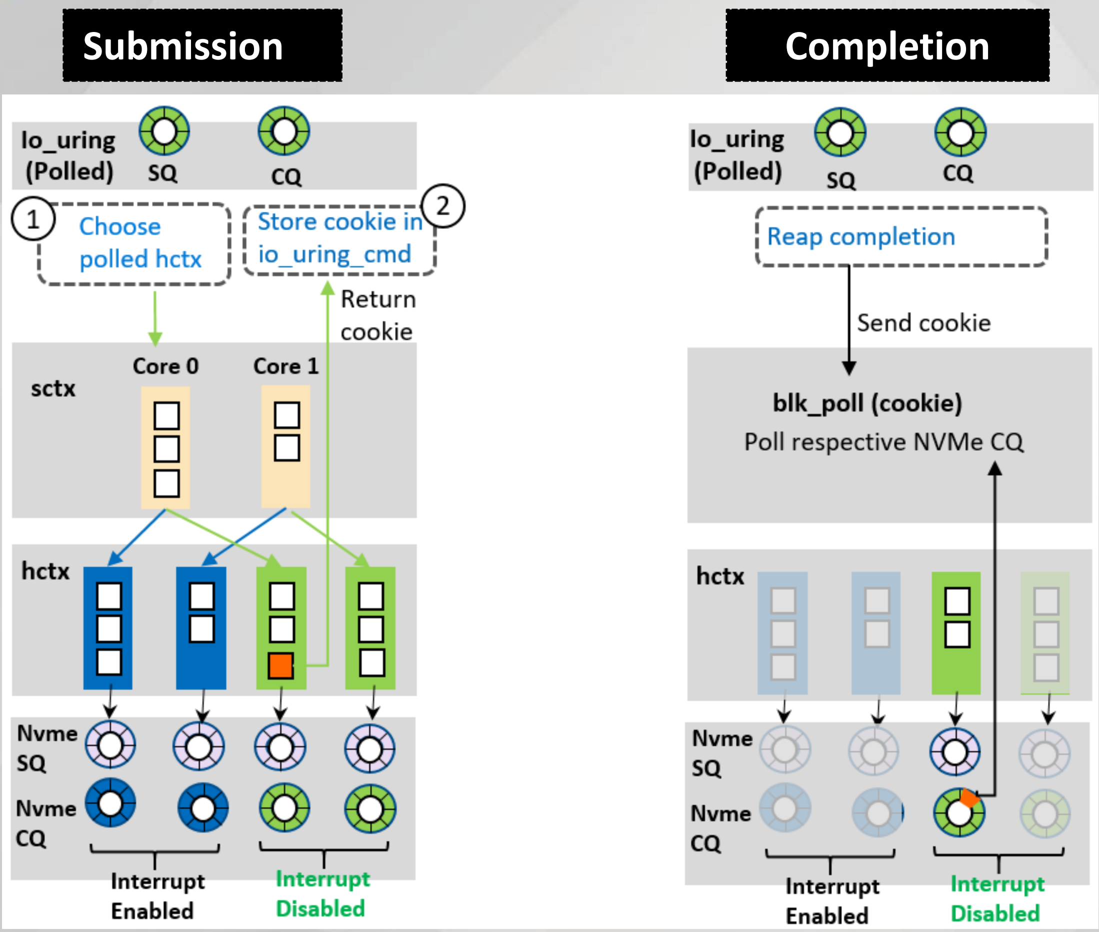
- NVMe plumbing
  - Instead of pin/unpin, talk to io_uring to reuse 'previously pinned' buffer/bio_vec
  - Same ioctl code; use uring_cmd info to choose between regular/fixed-buffer

```
int io_uring_cmd_import_fixed(void *ubuf, unsigned long len,
                              int rw, struct iov_iter *iter, void *ioucmd)
```

Kernel I/O Polling

- Enables interrupt-free IO; particularly useful for ultra-low-latency storage
- What we have
  - Sync polling: submit IO and spin for completion, *in the same syscall;* submit-spin
    - Preadv2()/pwritev2() with RWF_HIPRI
  - Hybrid polling – relax CPU by sleeping in between; submit-sleep-spin
  - Async polling: decouple polling from submission; provides third choice (beyond spin and sleep) i.e. submit more IO or execute app-specific logic
    - io_uring setup with IORING_SETUP_POLL; all IOs to such ring are polled
- What we do not have
  - ioctl polling / passthru polling

Uring passthru: async polling



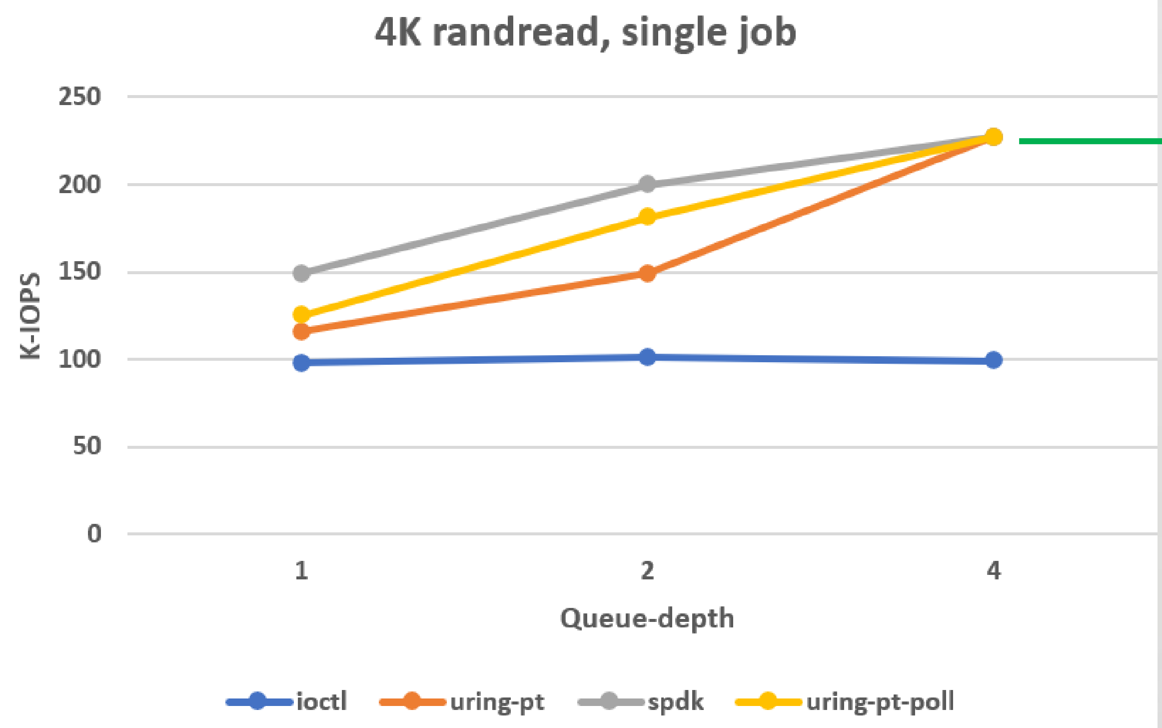
Features for faster IO

- Now this looks better than before

Feature	What it does	Io_uring	Uring-passthrough
Register-files	Reference fd once and reuse	☑	☑
SQPoll	Offload IO submission	☑	☑
Fixed-buffer	Map IO buffer once and reuse	☑	☑
Async polling	Interrupt-free completion	☑	☑
Bio-cache	In-kernel cache to reduce per-io alloc & free	☑	☒

- And there is new entry in the table: bio-cache
  - Recently merged
  - Not IRQ safe, so currently for polled-IO path
  - For NVMe Passthru – we almost always do in-task completion; so that sorts applicability issue
  - Passthru bio - currently allocated via bio_kmalloc() Move to bio-set based allocation for async path

How does it perform?



Device saturated at this point

LINUX PLUMBERS CONFERENCE › September 20-24, 2021

What is where

- NVMe Generic Device:
  - Kernel support: nvme 5.13
  - Nvme-cli: https://github.com/linux-nvme/nvme-cli/commit/7169d78c9ccc0835039dcb2ac6f48d4e697e5dcd
- Uring-cmd/async IOCTL:
  - Mailing list: https://lore.kernel.org/linux-nvme/20210317221027.366780-1-axboe@kernel.dk/
  - Refreshed version: https://git.kernel.dk/cgit/linux-block/log/?h=io_uring-fops.v6
- NVMe passthru
  - Async & fixed-buffer: https://lore.kernel.org/linux-nvme/20210805125539.66958-1-joshi.k@samsung.com/
  - Passthru polling: in due course, https://github.com/joshkan/nvme-uring-pt
  - Bio-cache: next step

Feedback

- Are there ideas to further optimize the path? (e.g. anything for DMA)