Linux Plumbers Conference

Dublin, Ireland September 12-14, 2022

Io_uring command
and
Modern NVMe passthrough

Where are we with the new I/O path: status and plans

Kanchan Joshi

Samsung Semiconductor India Research

Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

Acknowledgements

- First things first: credit where it’s due
  - Jens, Christoph, Stefan
  - Many other reviewers from io_uring and nvme list
  - LSM coverage: Luis, Paul Moore, Casey

Outline

- Why
  - Semantic gap between NVMe and Linux
  - How existing passthrough does not help
- What is cemented
  - Io_uring command: architecture
  - New nvme passthrough: design and performance
  - User-space outreach
- Discussion (on underway/missing pieces)

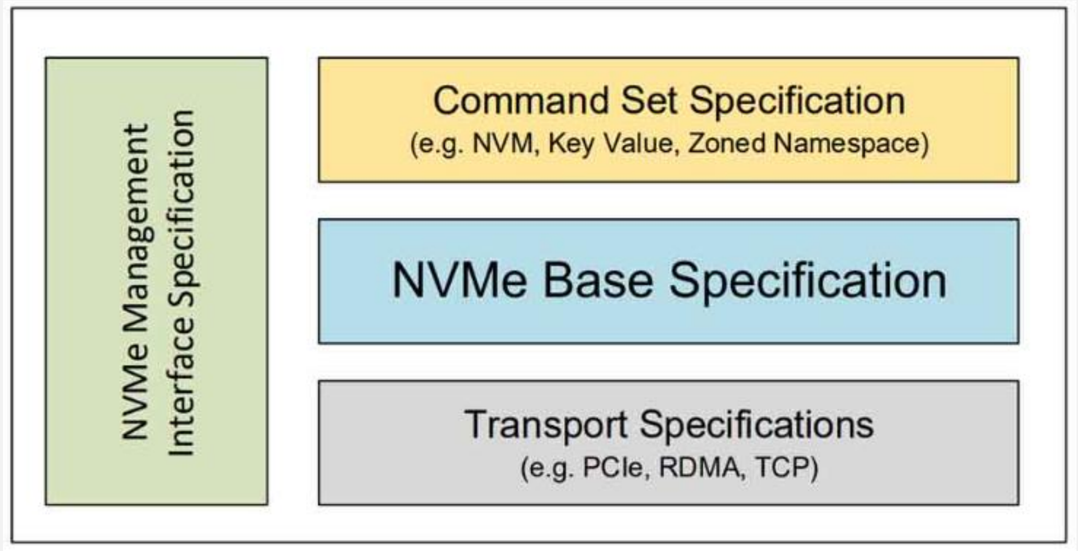
Why

Background and problem-statement

The semantic gap

NVMe Storage

- Rapid growth of new storage interfaces
  - New commands
    - Directives (streams), Copy (in-device)
  - New command sets
    - ZNS, KV, Computational storage (down the line)



- Require close collaboration with the Host
    - Predictable latencies, higher endurance
    - Reduced CPU/energy consumption

Linux Kernel

- Generic abstractions
  - Pro: Help dealing with a variety of devices in the same fashion
  - Con: the semantic gap between device and application interface. Emerging interfaces may not fit well within existing OS abstractions (e.g. POSIX)
- Novelty vs Maintenance
  - Can evolving/short-lived interfaces become a long-term maintenance burden
  - Can early technology adopters use the upstream kernel

Growing gap

NVMe Storage

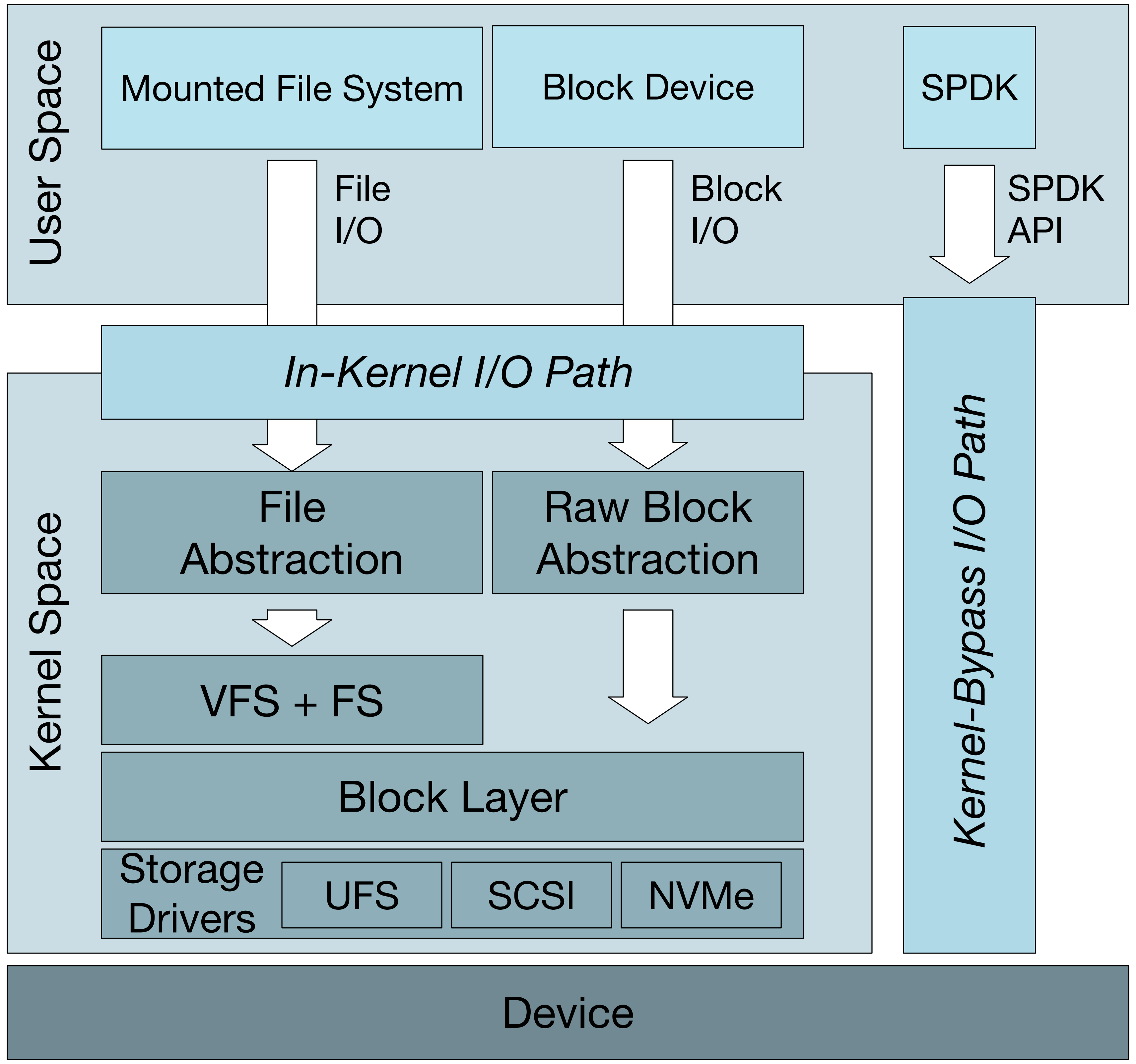
- I/O is no longer just ‘classical’ read/write
  - New constructs continue to emerge
    - Zone Append: late binding of written LBA
    - Copy-command: composite read + write
    - Store Keys, Retrieve Values (no concept of LBA)

Linux Kernel

- More friendly to ‘classical’ read/write
  - New ‘generic’ syscalls are hard to grow
  - If the interface can’t fit, it gets punted to ioctl
  - Ioctl: far from all the OS-level advancements that have gone into read/write syscalls

Read/Write (Direct IO)	Ioctl (nvme passthrough)
Async	Sync
Syscall-free submission (Submission Polling)	😢
Interrupt-free IO (Completion Polling)	😢
Vectored (multi-buffer)	😢
Registered file (Reuse open handle across multiple I/Os)	😢
Registered buffer (Reuse mapped buffer across I/Os)	😢

Existing storage I/O paths

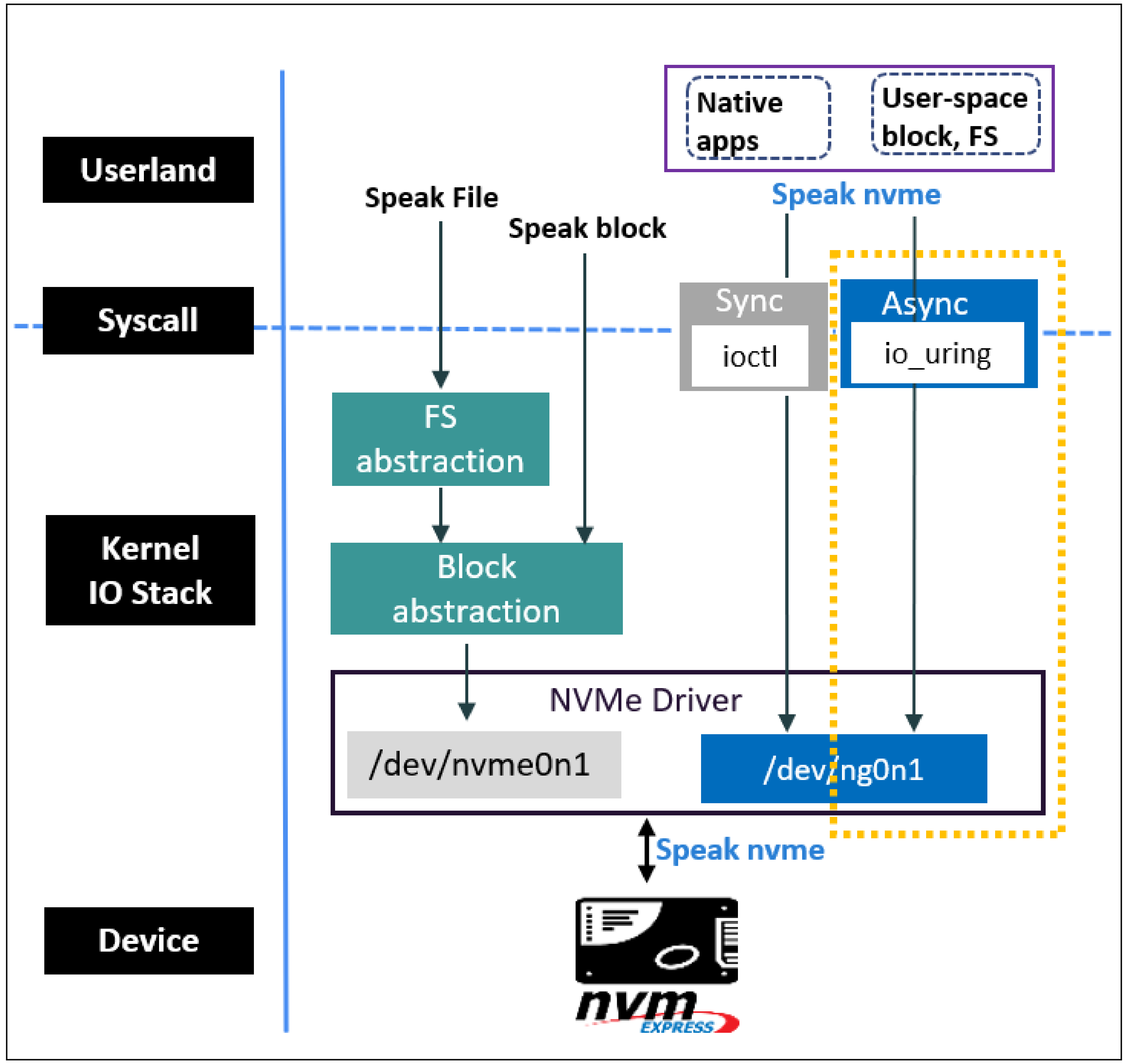


- Filesystem IO path
  - Prioritize stability/robustness over the new features (rightly so)
  - Prefer established technology vs cutting-edge features
- Block IO path
  - Conditional: not usable (zero-capacity, hidden, read-only etc.) if a device does not fit into block-abstraction or contains an unsupported feature
  - New feature, even if supported (via generic block command), requires a user interface. Otherwise, it gets punted to ioctl-driven passthrough
- SPDK IO path
  - User-space driven; supports fast innovation
  - Domain-specific, rather than generic

What

The new I/O path is all about, and how it helps

New catch-all fast path to NVMe



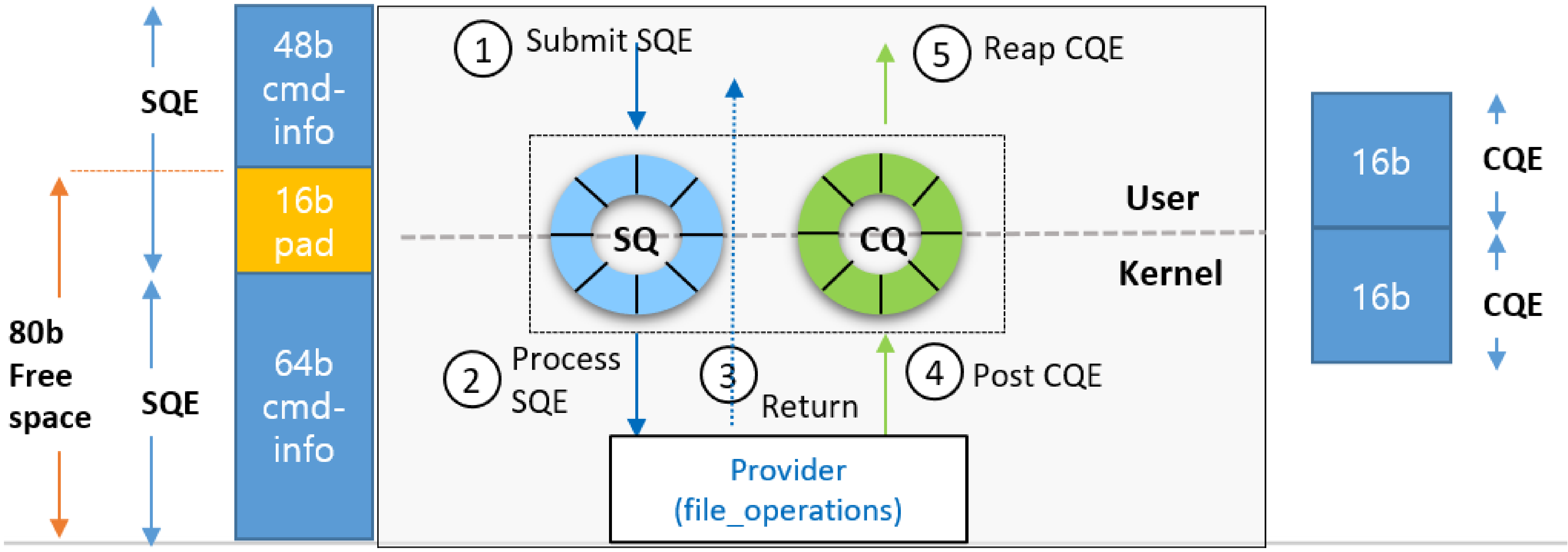
- NVMe generic char interface
  - Solves availability problem
  - Always comes up regardless of unsupported features or current/future command-sets
  - Nvme-native passthrough: same syscall for any nvme command
  - Agility to embrace new technology

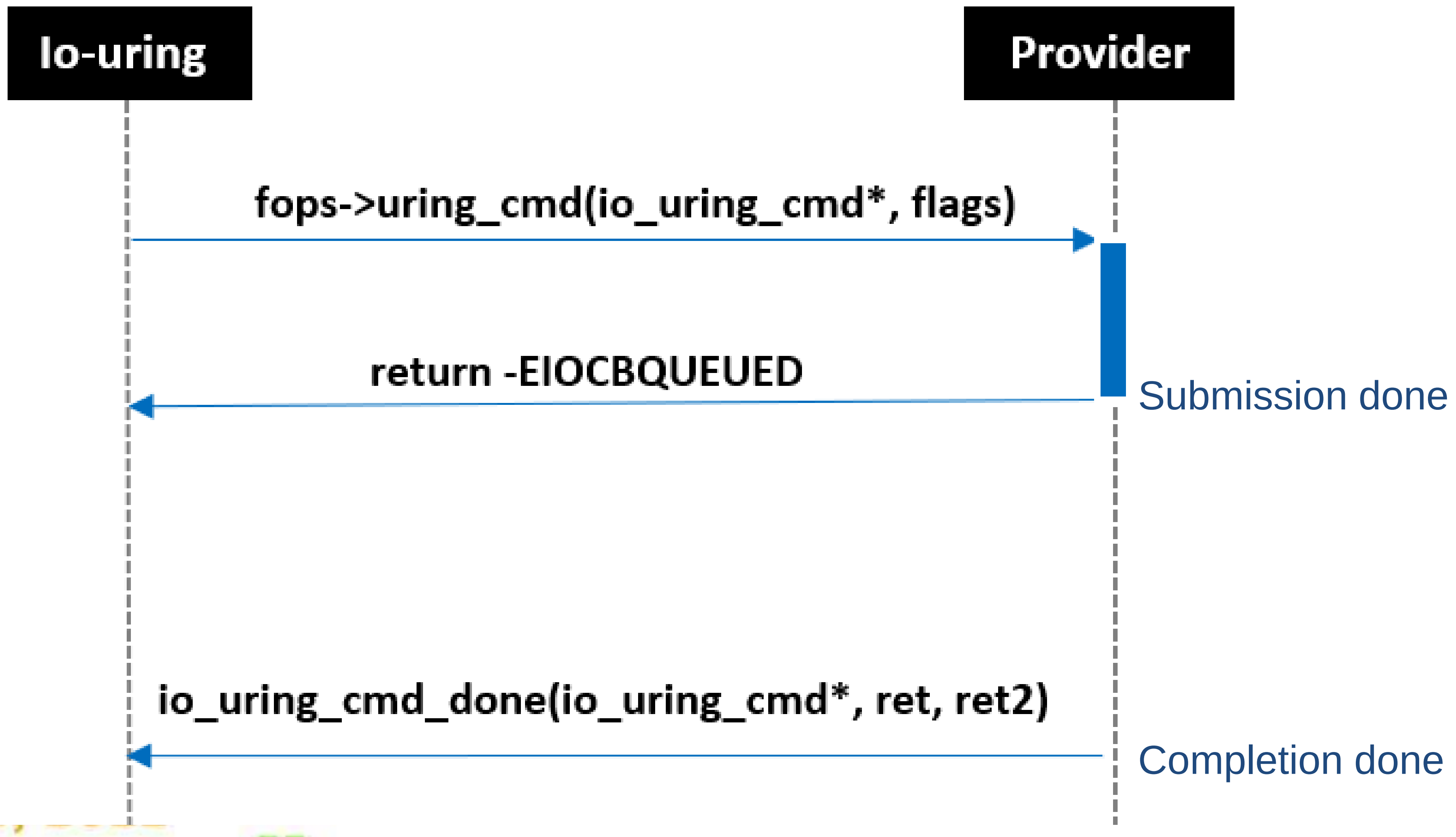
- Io_uring driven passthrough
  - Solves scalability problem
  - Attaches various io_uring capabilities to any nvme command

io_uring command

- Generic (not nvme) facility to attach io_uring capabilities for the underlying command
  - Co-work with command provider (driver, FS etc.); NVMe driver (from 5.19) and ublk (from 6.0)

- User interface
  - New opcode: IORING_OP_URING_CMD
  - Provider specific opcode: SQE->cmd_op
  - Place command inline in free space inside SQE; 16 bytes in regular SQE, 80 bytes in Big SQE
  - Result to arrive in CQE
    - one result into CQE->res as usual
    - Auxiliary result into Big CQE





Big SQE and Big CQE

- Double the size of regular SQE (128b)
  - Setup ring with the flag IORING_SETUP_SQE128

```
@@ -22,6 +22,7 @@ struct io_uring_sqe {
        union {
                __u64   off;    /* offset into file */
                __u64   addr2;
+               __u32   cmd_op;
        };
        union {
                __u64   addr;   /* pointer to buffer or iovecs */
@@ -61,14 +62,17 @@ struct io_uring_sqe {
                __s32   splice_fd_in;
                __u32   file_index;
        };
-       __u64   addr3;
-       __u64   __pad2[1];
-
-       /*
-        * If the ring is initialized with IORING_SETUP_SQE128, then this field
-        * contains 64-bytes of padding, doubling the size of the SQE.
-        */
-       __u64   __big_sqe_pad[0];
+       union {
+               struct {
+                       __u64   addr3;
+                       __u64   __pad2[1];
+               };
+               /*
+                * If the ring is initialized with IORING_SETUP_SQE128, then
+                * this field is used for 80 bytes of arbitrary command data
+                */
+               __u8    cmd[0];
+       };
 };
```

Provider-specific opcode (`__u32 cmd_op;`)

Inline cmd (starting offset) (`__u8 cmd[0];`)

- Double the size of regular CQE (64b)
  - Setup ring with the flag IORING_SETUP_CQE128

```
@@ -245,6 +246,12 @@ struct io_uring_cqe {
        __u64   user_data;      /* sqe->data submission passed back */
        __s32   res;            /* result code for this event */
        __u32   flags;
+
+       /*
+        * If the ring is initialized with IORING_SETUP_CQE32, then this field
+        * contains 16-bytes of padding, doubling the size of the CQE.
+        */
+       __u64 big_cqe[];
 };
```

16 more bytes to return extra result (`__u64 big_cqe[];`)

Ioctl-driven NVMe Passthrough

- Userland prepares “struct nvme_passthru_cmd64” (80 bytes) and sends ioctl with opcode NVME_IOCTL_IO64_CMD

```
#define NVME_IOCTL_ADMIN64_CMD   _IOWR('N', 0x47, struct nvme_passthru_cmd64)
#define NVME_IOCTL_IO64_CMD      _IOWR('N', 0x48, struct nvme_passthru_cmd64)
```

```
struct nvme_passthru_cmd64 {
        __u8    opcode;
        __u8    flags;
        __u16   rsvd1;
        __u32   nsid;
        __u32   cdw2;
        __u32   cdw3;
        __u64   metadata;
        __u64   addr;
        __u32   metadata_len;
        union {
                __u32   data_len; /* for non-vectored io */
                __u32   vec_cnt; /* for vectored io */
        };
        __u32   cdw10;
        __u32   cdw11;
        __u32   cdw12;
        __u32   cdw13;
        __u32   cdw14;
        __u32   cdw15;
        __u32   timeout_ms;
        __u32   rsvd2;
        __u64   result;
};
```

- Submission: Copy command from userspace to Kernel
- Completion: Copy result back to userspace

```
if (copy_from_user(&cmd, ucmd, sizeof(cmd)))
        return -EFAULT;
```

```
if (put_user(cmd.result, &ucmd->result))
        return -EFAULT;
```

io_uring driven nvme passthru

- Prepare new "struct nvme_uring_cmd" and specify new opcodes in "sqe->cmd_op"

```
/* io_uring async commands: */
#define NVME_URING_CMD_IO        _IOWR('N', 0x80, struct nvme_uring_cmd)
#define NVME_URING_CMD_IO_VEC    _IOWR('N', 0x81, struct nvme_uring_cmd)
#define NVME_URING_CMD_ADMIN     _IOWR('N', 0x82, struct nvme_uring_cmd)
#define NVME_URING_CMD_ADMIN_VEC _IOWR('N', 0x83, struct nvme_uring_cmd)
```

```
/* same as struct nvme_passthru_cmd64, minus the 8b result field */
struct nvme_uring_cmd {
        __u8    opcode;
        __u8    flags;
        __u16   rsvd1;
        __u32   nsid;
        __u32   cdw2;
        __u32   cdw3;
        __u64   metadata;
        __u64   addr;
        __u32   metadata_len;
        __u32   data_len;
        __u32   cdw10;
        __u32   cdw11;
        __u32   cdw12;
        __u32   cdw13;
        __u32   cdw14;
        __u32   cdw15;
        __u32   timeout_ms;
        __u32   rsvd2;
};
```

- Zero-copy between user/kernel
  - Submission: no copy_from_user (use Big SQE)
  - Completion: no put_user (use Big CQE)

```
struct io_uring_cmd {
        struct file     *file;
        const void      *cmd;
        /* callback to defer completions to task context */
        void (*task_work_cb)(struct io_uring_cmd *cmd);
        u32             cmd_op;
        u32             pad;
        u8              pdu[32]; /* available inline for free use */
};
```

- Zero fast-path allocations
  - Reuse pre-allocated memory for any bookkeeping

Read using uring-passthrough

First things first: use generic-char dev

Ask big SQE and big CQE (efficiency)

Arm the SQE with uring-command op

NVMe io/admin opcodes
- URING_CMD_IO/IO_VEC
- URING_CMD_ADMIN/ADMIN_VEC

Extract command from SQE (no allocation)

Populate NVMe command

Submit SQE

Reap completion, and get auxiliary result too

```
/* issue passthru command to read from device into buf */
void nvme_uring_cmd(void *buf)
{
        struct io_uring ring;
        struct io_uring_sqe *sqe = NULL;
        struct io_uring_cqe *cqe = NULL;
        struct nvme_uring_cmd *cmd;
        struct io_uring_params p = { };
        int fd;

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

        p.flags = IORING_SETUP_SQE128;
        p.flags |= IORING_SETUP_CQE32;
        io_uring_queue_init(1, &ring, p.flags);

        sqe = io_uring_get_sqe(&ring);
        sqe->fd = fd;
        sqe->opcode = IORING_OP_URING_CMD;
        sqe->user_data = 0x1234;

        sqe->cmd_op = NVME_URING_CMD_IO;
        cmd = (struct nvme_uring_cmd *)&sqe->cmd;
        prepare_pt_cmd(cmd, buf);

        io_uring_submit(&ring);

        io_uring_wait_cqe(&ring, &cqe);
        __s32 status = cqe->res;
        __s64 result1 = cqe->big_cqe[0];
        printf("%s status:%d result1:%lld\n", __func__, status, result1);
        io_uring_cqe_seen(&ring, cqe);
        io_uring_queue_exit(&ring);
}
```

Upstream status

- NVMe Generic device
  - Initial support: 5.13 (June 2021)
  - Anonymous command-set: 6.0

- Passthrough path
  - Io_uring cmd: 5.19 (July 2022)
  - New passthrough for nvme: 5.19
  - Uring-cmd-poll: scheduled for 6.1

Read/Write (Direct IO)	Ioctl-nvme-passthru	Uring-nvme-passthru
Async	Sync	Async
Syscall-free submission (Submission Polling)	☒	☑
Interrupt-free IO (Completion Polling)	☒	☑
Vectored (multi-buffer)	☒	☑
Registered file (Reuse open handle across multiple I/Os)	☒	☑
Registered buffer (Reuse mapped buffer across I/Os)	☒	v7

User-space support and tooling

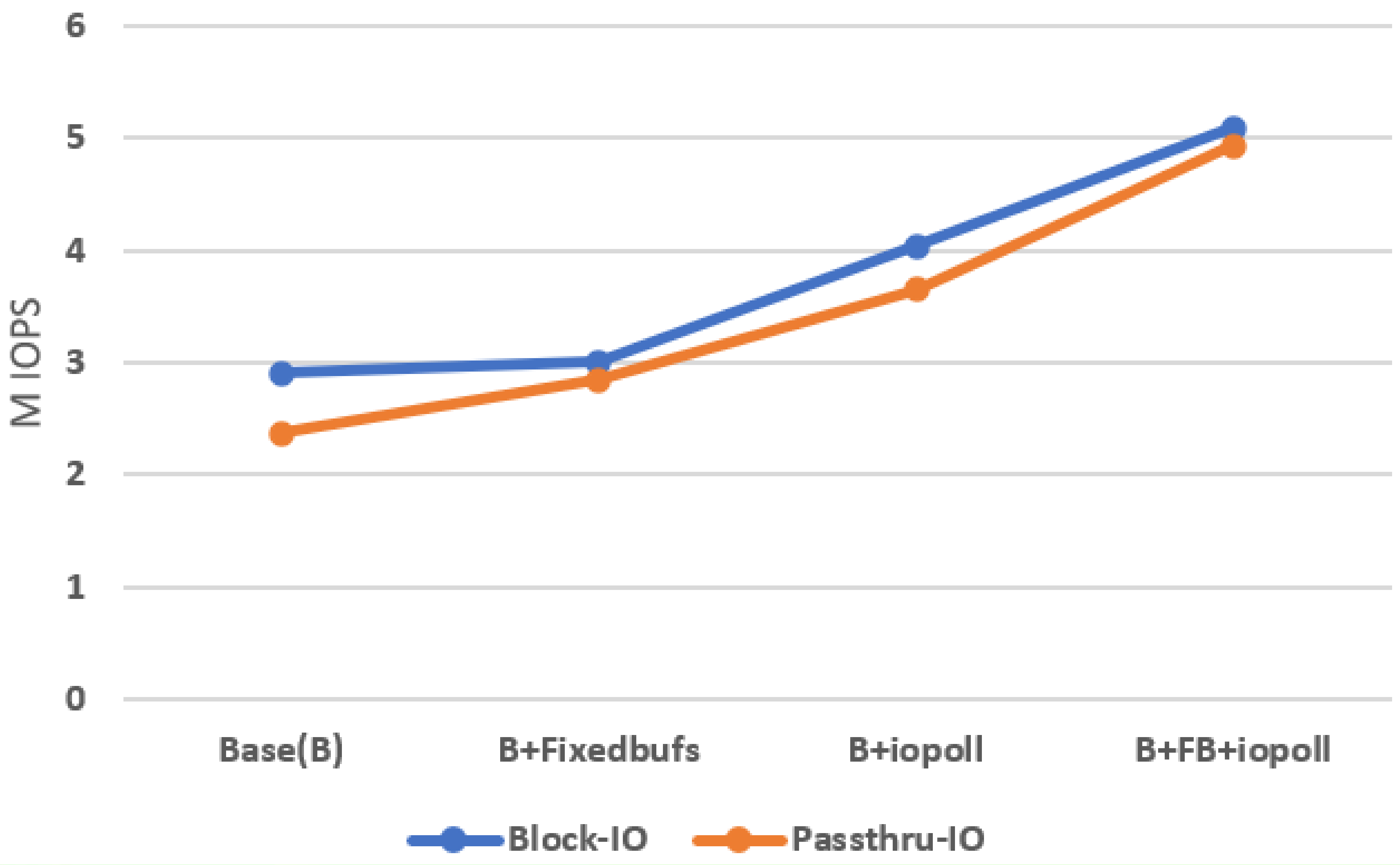
- xNVMe [1]: new backend for passthru/io_uring_cmd
- SPDK: new Bdev that understands io_uring_cmd; upcoming in 22.09 release
  - https://github.com/spdk/spdk/commit/6f338d4bf3a8a91b7abe377a605a321ea2b05bf7
- Ublk user-space: uses io_uring cmd, but not the nvme parts
- Libblkio: block device I/O library. Uses nvme-passthrough. C and RUST binding too [2]

- Nvme-cli: can list and operate on /dev/ngXnY
- Fio: new io engine for io_uring_cmd; Peak-perf test (t/io_uring) support
- Liburing: new tests“test/io_uring_passthrough.t

[1] I/O interface independence with xNVMe: https://dl.acm.org/doi/10.1145/3534056.3534936
[2] https://gitlab.com/libblkio/libblkio

How does it scale?

- Borrowed from Jens (since my setup shows passthru doing bit better than the block and I can't believe it)
- Peak performance test, Optane Gen 2
  - t/io_uring -b512 -d128 -c32 -s32 -p0 -F1 -B0 -O0 -P1 -u1 -n1 /dev/ng0n1



512b RR	Block-IO	Passthru-IO
Base(B)	2.9	2.37
B+Fixedbufs	3	2.84
B+iopoll	4.04	3.65
B+FB+iopoll	5.09	4.93

- Passthru: absence of batched tag free/allocation

Discussion

& further work items

NVMe: max IO size limit

- Device will have a limit on how large a single IO can be. But Driver also has its own limit
- IO with the size 512KB (>4K * 127) fails often; Due to memory fragmentation. Bit ugly on a device that can support >= 2MB single IO
- Block-path does not face it as IO splitting is done by block-layer
- Current solution: Application should use hugepage backed allocation
- Anything better than that? Something that can support 4MB limit

```
/*
 * These can be higher, but we need to ensure that any command doesn't
 * require an sg allocation that needs more than a page of data.
 */
#define NVME_MAX_KB_SZ   4096
#define NVME_MAX_SEGS    127
```

4MB limit

Much smaller limit

```
/*
 * Double check that our mempool alloc size will cover the biggest
 * command we support.
 */
alloc_size = nvme_pci_iod_alloc_size();
WARN_ON_ONCE(alloc_size > PAGE_SIZE);

dev->iod_mempool = mempool_create_node(1, mempool_kmalloc,
                                        mempool_kfree,
                                        (void *) alloc_size,
                                        GFP_KERNEL, node);
```

nvme-whitelisting

- NVMe driver keeps io/admin commands CAP_SYS_ADMIN check, with no regard to file permission bits

```
$ ls -l /dev/ng*
crw-rw-rw- 1 root root 242, 0 Sep  9 19:20 /dev/ng0n1
crw------- 1 root root 242, 1 Sep  9 19:20 /dev/ng0n2
```

ng0n1 appears to be allowing unprivileged read/write access, but it does not

- Nvme-whitelist (similar to SCSI)
  - Move from blanket CAP_SYS_ADMIN to fine-grained control as per file-handle permission
  - Should we consider whitelisting few safe read-only admin-cmd (e.g. identify) that give necessary info for forming io-command (e.g. lba format, namespace capacity)

```
 * Only a subset of commands are allowed for unprivileged users. Commands used
 * to format the media, update the firmware, etc. are not permitted.
 */
bool scsi_cmd_allowed(unsigned char *cmd, fmode_t mode)
{
        /* root can do any command. */
        if (capable(CAP_SYS_RAWIO))
                return true;

         case ZBC_IN:                                    a read-safe command */
                 return true;
         /* Basic writing commands */
         case WRITE_6:
         case WRITE_10:
         case WRITE_VERIFY:
         case WRITE_12:
         case WRITE_VERIFY_12:
         case WRITE_16:
                return (mode & FMODE_WRITE);
```

NVMe multipathing

- Enterprise NVMe SSDs may have dual controllers that help in implementing HA
- CONFIG_NVME_MULTIPATH
  - nvme driver keeps multipathing (failover, requeue) abstracted from user-space
  - That is for block path
- Passthrough path
  - Current policy: Return failure to userspace so that it can retry the IO on an alternate path
  - Or we go about implementing failover/requeue for passthrough IO [1]
    - Queuing io_uring_cmd (as opposed to bio) was not clean
    - And SQE lifetime (submission-only) caused some churn too

[1] https://lore.kernel.org/linux-nvme/20220711110155.649153-1-joshi.k@samsung.com/

LSM for uring-cmd

- Traditional Linux security model is DAC based (root/user/groups/read-write-execute permissions)
- But we also have MAC security model - multiple LSMs implementing MAC (e.g. SELINUX, Smack, Apparmor)
- LSM for uring-cmd:
  - 5.19 did not have LSM support for uring-cmd
  - 6.0 has - SELINUX and Smack hooks. And this is marked to be backported for 5.19 too
- Are there things that we still are missing?
  - Ioctl opcode vs SQE->cmd_op

```
/* io_uring async commands: */
#define NVME_URING_CMD_IO        _IOWR('N', 0x80, struct nvme_uring_cmd)
```

    - 32bit ioctl opcode: 2 bits (direction) + 8 bits (type) + 8 bits (number for the type) + 14 (size of argument)
    - This gives more information to LSM to be fine-granular in its decision-making (i.e. reject less often?)
    - For SQE->cmd_op we do not have the format enforced.

Towards more efficiency

- DMA pre-mapping support is under discussion. Keith's patches [1]
- One of the discussion point: requiring new bio type, and corresponding changes in block path
- For passthrough path: DMA cookie goes into io_uring_cmd, and we should be able to skip creating bio
- Now something more imaginary than real (and if all goes well with the above)
  - Connecting nvme and io_uring more directly (both have SQ/CQ interface)
  - "direct_queues = X" (like poll_queues) and special ring in io_uring
  - We may just be able to avoid creating 'struct request', and core/queue mapping and tag-management can be part of io_uring ring management

[1] https://lore.kernel.org/linux-nvme/20220809064613.GA9040@lst.de/

Thanks