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
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

- 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

- 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
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)

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

### NVMe Storage

<table>
<thead>
<tr>
<th></th>
<th>Read/Write (Direct IO)</th>
<th>ioctl (nvme passthrough)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Async</td>
<td></td>
<td>Sync</td>
</tr>
<tr>
<td>Syscall-free submission (Submission Polling)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt-free IO (Completion Polling)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectored (multi-buffer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registered file (Reuse open handle across multiple I/Os)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registered buffer (Reuse mapped buffer across I/Os)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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

---

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

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

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

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

- Submission: Copy command from userspace to Kernel
- Completion: Copy result back to userspace
• Prepare new “struct nvme_uring_cmd” and specify new opcodes in “sqe->cmd_op”

```c
/* 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)
```

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

```c
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/I0_VEC
- URING_CMD_ADMIN/ADMIN_VEC

Extract command from SQE (no allocation)

Populate NVMe command

Submit SQE

Reap completion, and get auxiliary result too
### 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

<table>
<thead>
<tr>
<th>Read/Write (Direct IO)</th>
<th>ioctl-nvme-passthru</th>
<th>Uring-nvme-passthru</th>
</tr>
</thead>
<tbody>
<tr>
<td>Async</td>
<td>Sync</td>
<td>Async</td>
</tr>
<tr>
<td>Syscall-free submission (Submission Polling)</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Interrupt-free IO (Completion Polling)</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Vectored (multi-buffer)</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Registered file (Reuse open handle across multiple I/Os)</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Registered buffer (Reuse mapped buffer across I/Os)</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Linux Plumbers Conference**

Dublin, Ireland  Sept. 12-14, 2022
User-space support and tooling

- **xNVMe** [1]: new backend for passthrus/io_uring_cmd
- **SPDK**: new Bdev that understands io_uring_cmd; upcoming in 22.09 release
  - [https://github.com/spdk/spdk/commit/6f338d4bf3a8a91b7abe377a605a321ea2b05bf7](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"

[2] [https://gitlab.com/libblkio/libblkio](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

<table>
<thead>
<tr>
<th>512b RR</th>
<th>Block-IO</th>
<th>Passthru-IO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base(B)</td>
<td>2.9</td>
<td>2.37</td>
</tr>
<tr>
<td>B+Fixedbufs</td>
<td>3</td>
<td>2.84</td>
</tr>
<tr>
<td>B+iopoll</td>
<td>4.04</td>
<td>3.65</td>
</tr>
<tr>
<td>B+FB+iopoll</td>
<td>5.09</td>
<td>4.93</td>
</tr>
</tbody>
</table>

- 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.

```c
#define NVME_MAX_KB_SZ 4096
#define NVME_MAX_SEGS 127

/*
 * 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.
 */

allocator = nvme_pci_iodev_alloc_size();
WARN_ON_ONCE(allocator_size > PAGE_SIZE);

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

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

```bash
$ 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)

```c
* 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:
        return true;  /* a read-safe command */

    /* 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
    - 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