STORAGE DEVELOPER CONFERENCE

SDC 23

BY Developers FOR Developers

libvfn

A low-level NVMe Application and VFIO Driver Framework

Klaus Jensen, Samsung Electronics

What is **libvfn**?

- Two “libraries”
  - A **VFIO** utility library (`#include <vfn/vfio.h>`) with helpers for writing user space drivers for any PCI device
    - Core helpers – vfio configuration, device bring up, IRQ configuration, mmio
    - IOMMU helpers – iommu api, I/O virtual address allocator
  - An **NVMe** user space driver (`#include <vfn/nvme.h>`)
    - Polling and event-driven modes
    - Low-level queue and register API
- **LGPL**, **MIT** dual-licensed
  - Core library has zero external dependencies
  - libnvme (and some GPL licensed support libraries) required for building tests and examples
- Designed (for now) for **x86_64** and **ARM64**

Sigh, another user space driver? **Why**?

- io_uring, **io_uring_cmd**… **xNVMe**? Hello?
  - **io_uring_cmd** has dramatically reduced the need for user space NVMe drivers
    - io_uring_cmd allows user space to “talk shop” (sending raw-ish NVMe commands)
  - **xNVMe** provides high-performance abstractions over block (and raw NVMe I/O)
    - unified API supporting several backends
      - linux aio, io_uring, io_uring_cmd, spdk…, and **libvfn**
    - command submission helpers
    - callback-based “reactor” for completions
    - NVMe type definitions
    - asynchronous and synchronous submission modes

io_uring_cmd and NVMe

- Fundamentally this enables a user to
  - Submit **raw-ish** NVMe commands
    - The submitted payload is slightly different from NVMe
      - PRP1 repurposed as a single 64 bit pointer to a virtual memory address (or `struct iovec`)
      - PRP2 split into two 32 bit values describing length (*or number of vector elements*) of the metadata and data pointers

- … while not having to worry about bootstrapping the driver
  - enabling, probing namespaces, configuring queues, etc.

io_uring_cmd and NVMe

- Available NVMe lingo remains bound by the environment provided (and **optionally enforced**) by the kernel driver
  - Without CAP_SYS_ADMIN…
    - only I/O commands without ”dangerous” command effects.
    - simple white-listed admin commands (identify, etc.).
  - With CAP_SYS_ADMIN…
    - What if you delete queues? Detach a namespace? **Whelp**.
      - You can insmod garbage.ko at anytime anyway, so no biggie really.
    - A cardinal rule of using io_uring_cmd is **do not screw up the driver**.

But, **you want it all**

- As a host/device verification engineer, you want to
  - issue **any** command and observe the fallout
    - you probably do not care about the block layer, file systems, etc.
    - you want to issue malformed commands (invalid PRPs, SGLs)
  - without potentially (or rather, *likely*) bringing down the kernel with a fat finger resulting in an Oops and a reboot

- And **that** is the domain of the **safe** user space driver
  - It's not **just** for performance

User space drivers and NVMe

- Fundamentally this enables a user to
  - Submit raw (as in **totally raw**) NVMe commands
  - … in a safe way
    - no risk of breaking the kernel (there is no driver to break)
    - … you might brick the drive if not careful

- … while having to **write a driver** in user space to
  - bootstrap the controller (configuring admin queues, probing namespaces, setting up queue memory, handling PRPs/SGLs…)

libvfn vs. **The Current State of The Art**

- Other user space drivers
  - SPDK
  - PyNVMe
  - QEMU `block/nvme`

- A **direct comparison** is **not fair** to either parties
  - SPDK is **so much more** than just an NVMe driver
    - io_uring_cmd-based bdev_xnvme is **closing the gap** with lib/nvme
  - PyNVMe is a **test-dedicated** NVMe driver with a **native Python API**
    - To the best of my knowledge, derived from SPDK
    - Since v3, no longer open source

libvfn vs. **SPDK**

- libvfn has similarities to SPDK, why is this work not in SPDK?
  - SPDK is more than an NVMe driver, it is a “**development kit**”, an **application framework**
    - bdev, fabrics support, etc.
    - **lots of useful sugar**
  - libvfn is fundamentally a userspace **PCIe driver framework**
    - maybe more similarities with **DPDK**
    - a low-level NVMe driver is included because that is what spurred the ”libvfio” part
- libvfn might be more suitable of embedding into other projects (**YMMV**)
  - **Zero** dependencies
  - Supports both polling and event-driven modes out of the box
    - minimal API – **less sugar included**

QEMU NVMe driver

- The QEMU NVMe driver allows the QEMU block layer to use a PCIe NVMe device directly as the **underlying storage** of VMs.
  - an emulated device is layered **on top** (e.g., `virtio-blk` or **even `hw/nvme`**)
  - single I/O queue pair
  - **extremely "to the point"** when disregarding all the QEMU block layer plumbing
- libvfn borrows two techniques
  - **Fast** command tracking
  - Relatively **simple** IOVA allocator
- See **Fam Zheng** at KVM Forum '19
  https://www.youtube.com/watch?v=bwyHxb4tng0

What makes user space drivers tick (**safely**)?

- **Safe** user space PCIe device drivers rely on
  1. the presence of a **DMA remapping facility** (a Translation Agent, *or **TA***) to ensure isolation through **host-managed mappings** (in Address Translation and Protection Tables, *or **ATPTs***)
     - The PCI Express specification defines the concept (but not the implementation) of TAs
     - Intel VT-d, AMD-Vi and ARM SMMU implement and provide such facilities

  1. raw **register access** (typically memory-mapped I/O)

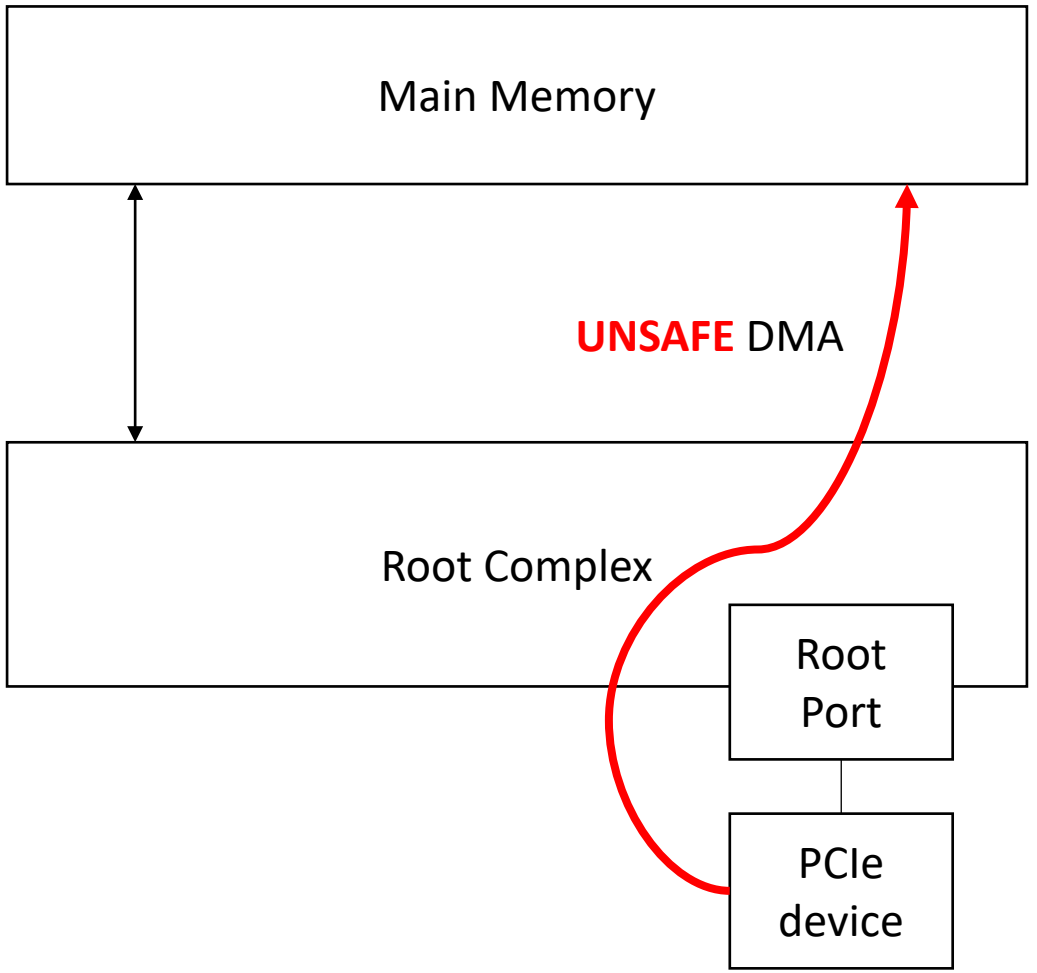
  2. and **interrupt programmability**

PCIe **Address Translation**

… and VFIO

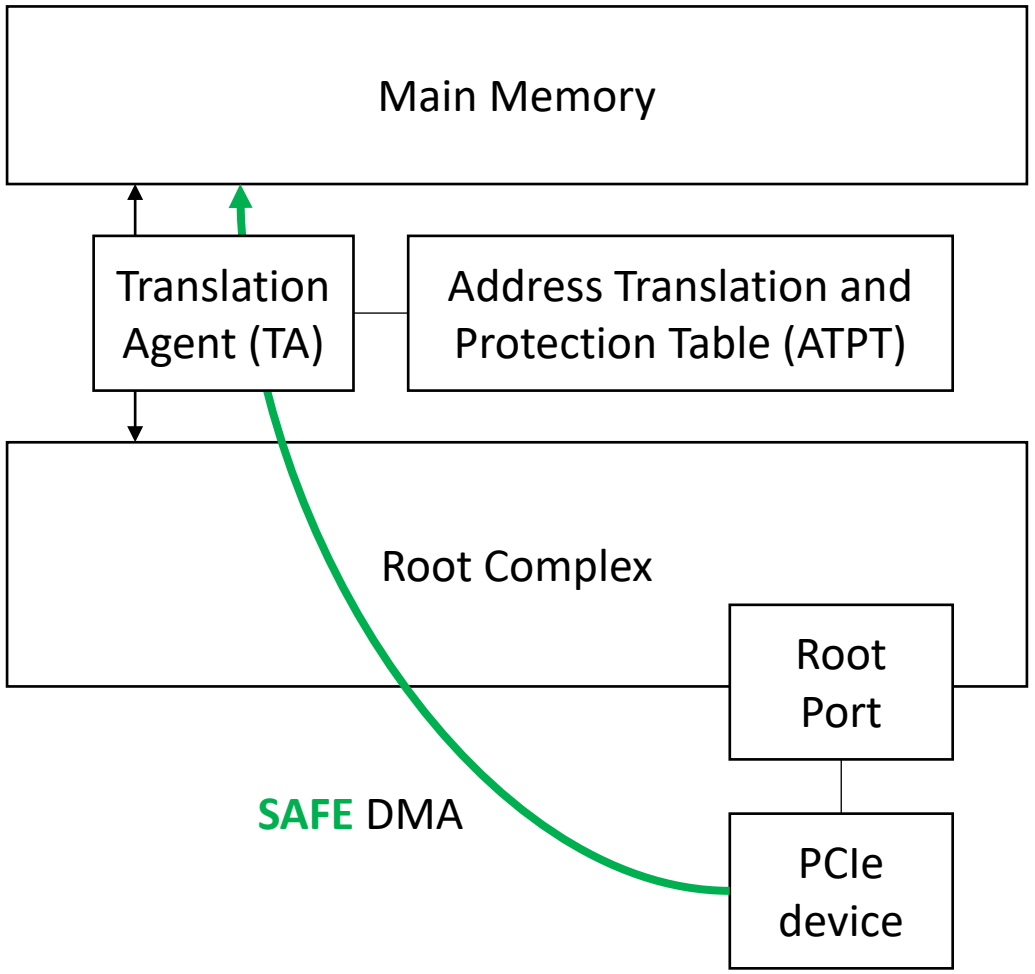
Legacy Platforms

- In general, on legacy platforms, PCI devices have **full access** to the **entire host physical memory address space**
  - ”entire address space” depends on device capabilities
    - if or not it can access 64 bit addresses
  - Maliciously (or not), hardware may exploit this



Modern Platforms

- On **modern** platforms, memory transactions **may** pass through a **Translation Agent**
  - **Software** (typically the OS) maintains a **translation table** which controls what physical addresses a device may access
    - Allows the OS to **protect itself** against faulty (or malicious) hardware, but **NOT** from buggy drivers



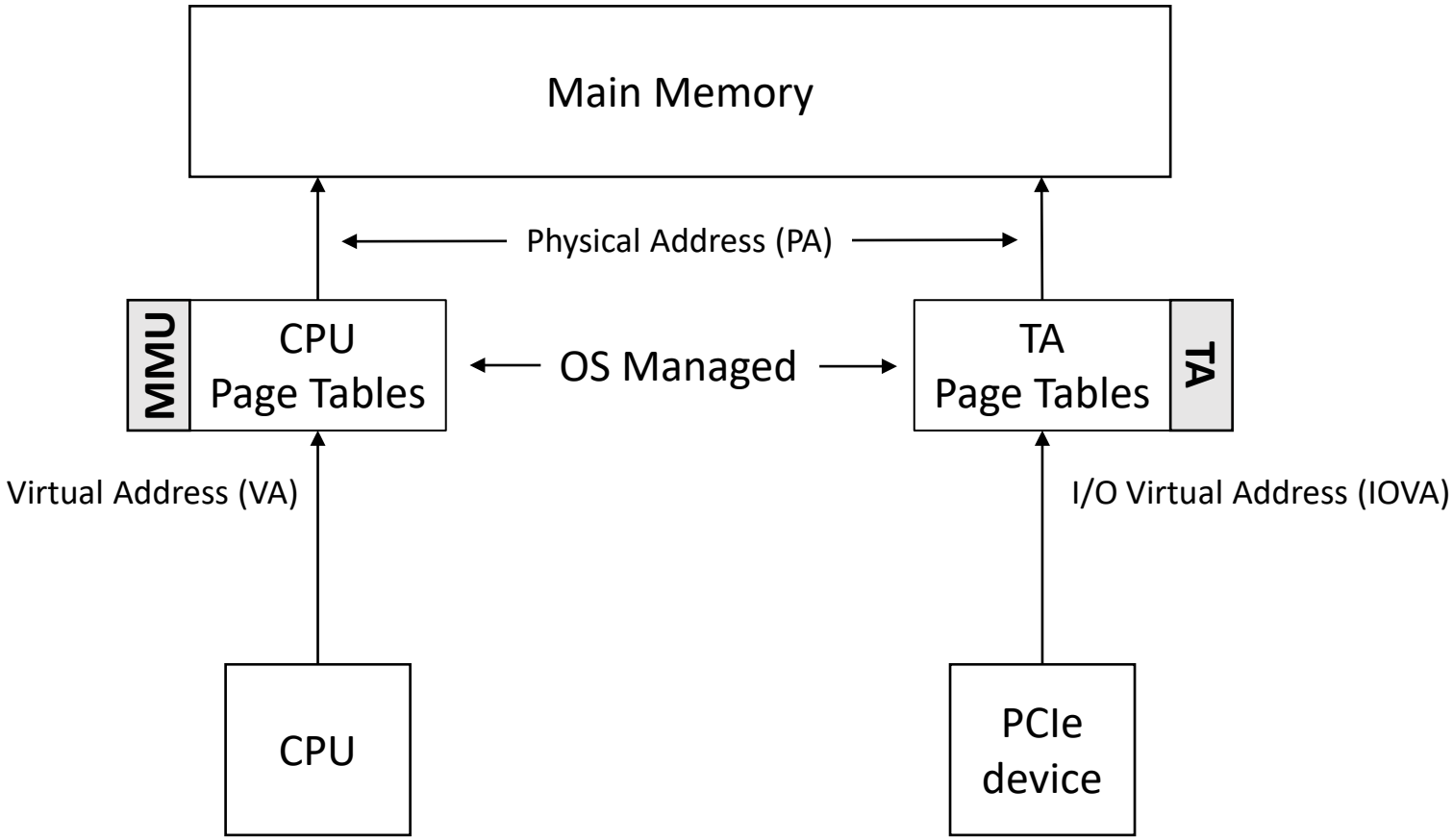
Not **just** for security…

Benefits of Address Translation

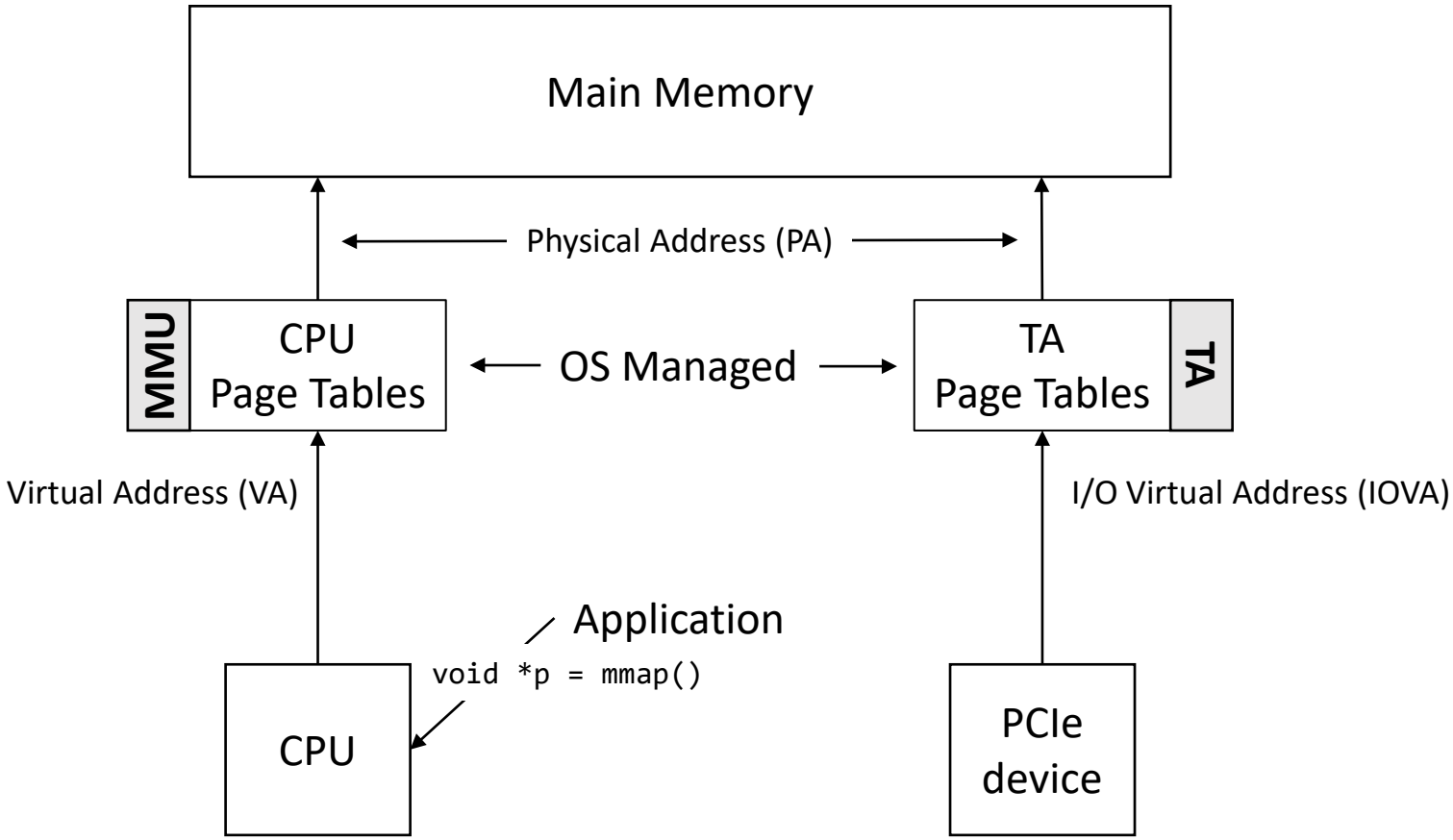
- Address **remapping**
  - map **discontiguous** memory into a **contiguous** range (scatter/gather)
  - allow **32-bit only devices** to access memory in **64-bit host memory** space
    - this was the original intent and purpose of a TA

- Enable **safe user-space drivers**
  - some smart guys figured out that this could be used by the kernel to
    - allow user-space to create mappings, but only for its own memory pages
    - isolate devices from each other

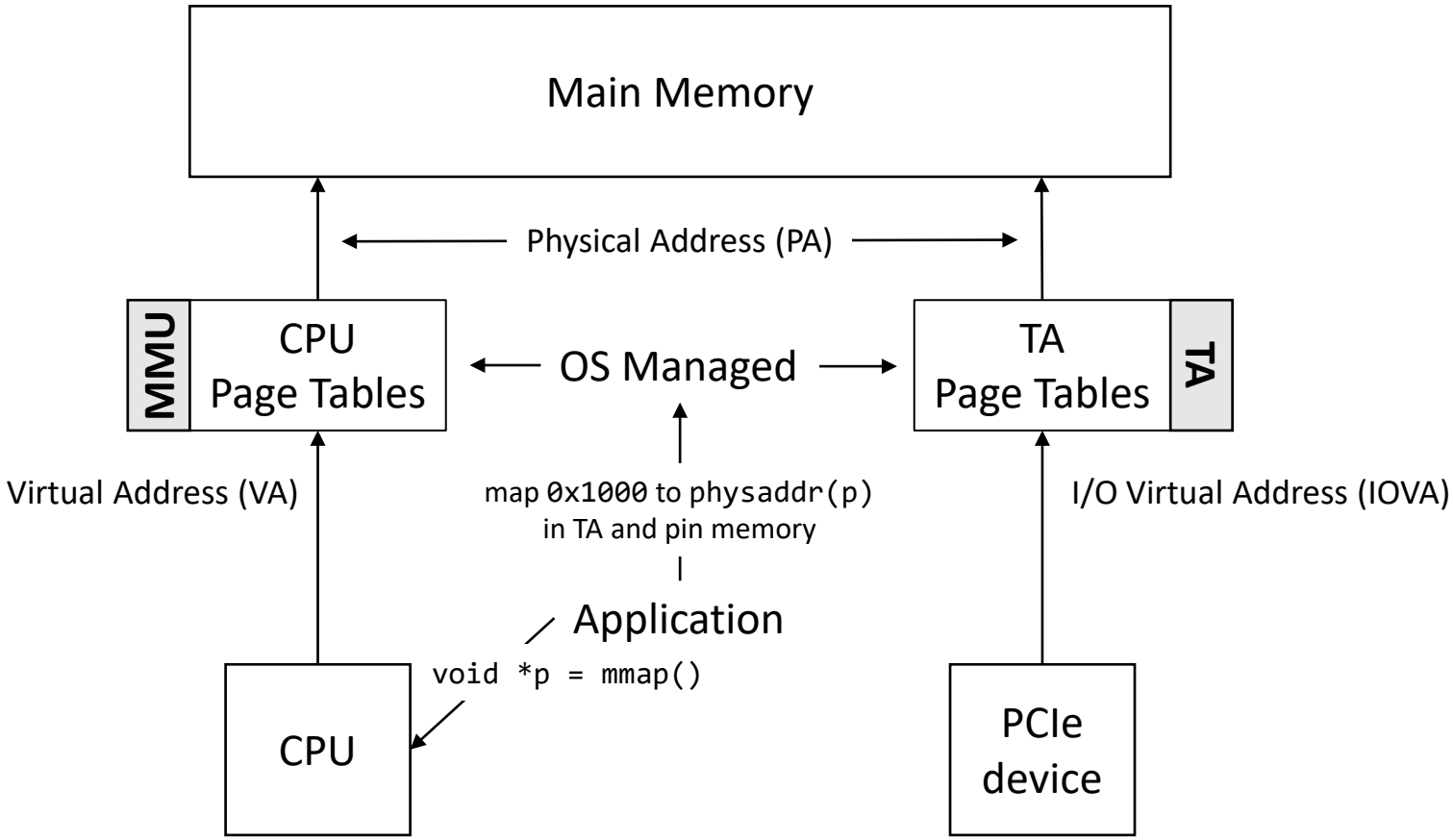
Typical Use



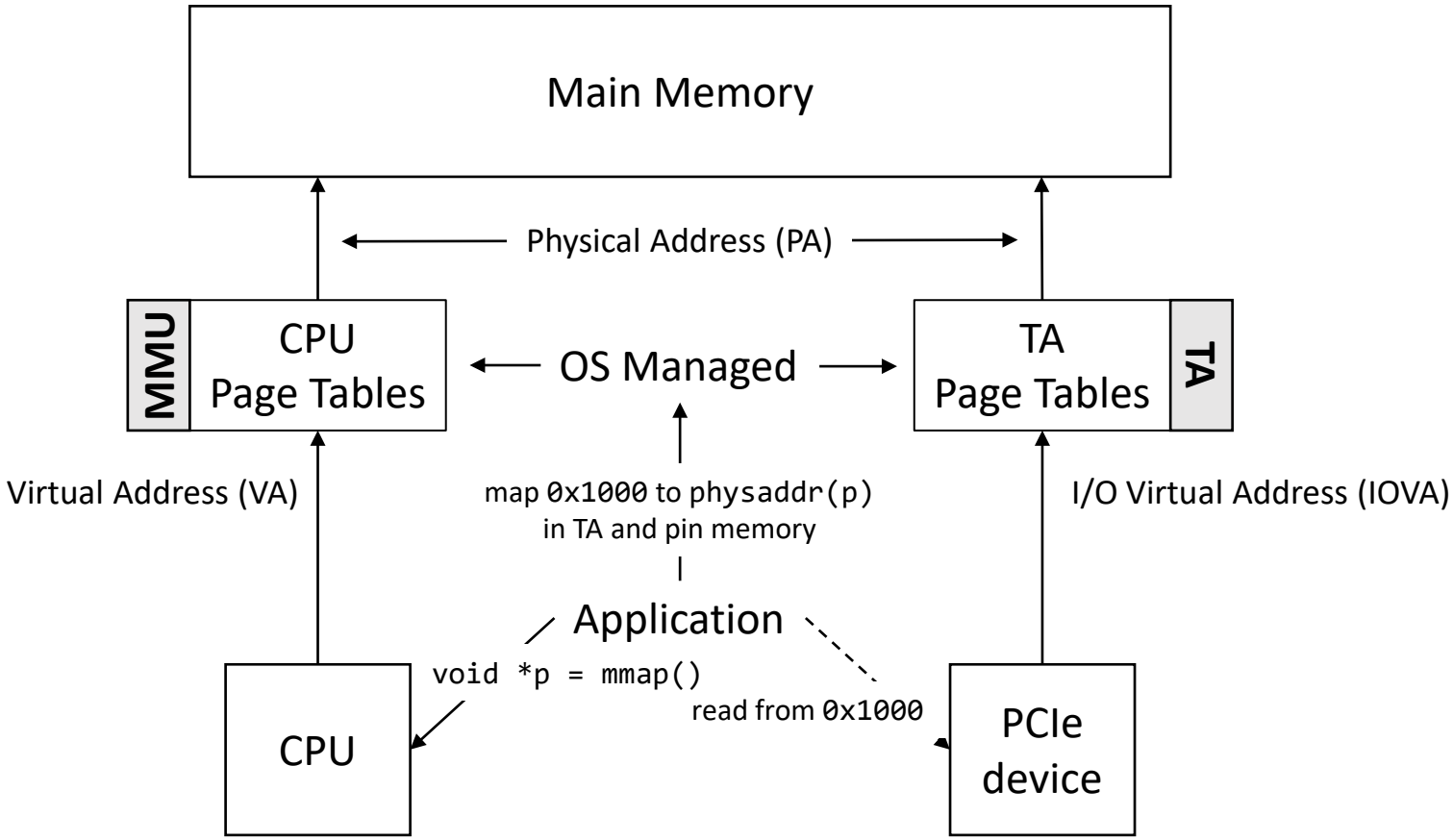
Typical Use



Typical Use



Typical Use



Translation Agent Implementations

- The PCI specification only defines the concept of the Translation Agent as a logical entity – **The details are vendor specific**

- The capabilities of the TA **varies**
- The format of the ATPT **varies**
- The PCI topology defines the **granularity** of isolation

- VFIO (*and now, IOMMUFD*) unifies it all under **common uAPIs**

A **unified** API

- An IOMMU and device **agnostic** API for **securely** exposing direct device access to userspace.
- **Three** main concepts - **Containers**, **Groups** and **Devices**
  - The **container** manages address translations (a set of page tables) for a set of **groups**
    - ioctls: `SET_IOMMU`, `IOMMU_MAP_DMA`, ...
  - The **group** represents a set of devices that share an isolation granularity
    - ioctls: `SET_CONTAINER`, `GET_DEVICE_FD`, ...
  - The **device** is a, …well, device
    - ioctls: `GET_REGION_INFO`, `SET_IRQS`, `RESET`, ...

Do we need a **libvfio**?

- Best practices for using VFIO is scattered amongst various projects
  - QEMU (`hw/vfio`, `util/vfio-helpers.c`)
  - DPDK (`lib/eal/linux/eal_vfio.{h,c}`)

- Verbose uAPIs
  - The uAPIs can be cumbersome, non-trivial and boiler-plate heavy to use
    - risk of mistakes, steep learning curve
  - Duplication, redundant code
    - duplication, redundant code

Using VFIO is a **little** boiler plate heavy

- Steps required to bring up a PCI device (configure **iommu group**)
  1. Configure VFIO “container” (open `/dev/vfio/vfio`)
     a. Verify API version
     b. Verify IOMMU support
  2. Configure IOMMU group
     a. Determine iommu group of device (i.e. `/dev/vfio/N`)
     b. Determine if iommu group is “viable”
     c. Set (attach) group to container
  3. Configure IOMMU
     a. Set IOMMU type on container
     b. Retrieve IOMMU information (capabilities)

Using VFIO is a **little** boiler plate heavy

- Steps required to bring up a PCI device (configure **device**)
  1. Get device handle (file descriptor)
  2. Get device information
     a. Verify that device is a PCI device
     b. Get device region information (PCI configuration space)
  3. Configure, initialize BARs
     a. Get device region information per BAR
  4. Set PCI bus master (write to configuration space)
  5. Configure IRQs
     1. Determine IRQ mechanisms (INTx, MSI, MSI-X)
     2. Select IRQ mechanism depending on support

Using VFIO is a **little** boiler plate heavy

```
/* Required data structures */
int container, group, device, i;
struct vfio_group_status group_status = { .argsz = sizeof(group_status) };
struct vfio_iommu_type1_info iommu_info = { .argsz = sizeof(iommu_info) };
struct vfio_iommu_type1_dma_map dma_map = { .argsz = sizeof(dma_map) };
struct vfio_device_info device_info = { .argsz = sizeof(device_info) };

/* Create a new container */
container = open("/dev/vfio/vfio", O_RDWR);

if (ioctl(container, VFIO_GET_API_VERSION) != VFIO_API_VERSION)
  /* Unknown API version */

if (!ioctl(container, VFIO_CHECK_EXTENSION, VFIO_TYPE1_IOMMU))
  /* Doesn't support the IOMMU driver we want. */

/* Open the group */
group = open("/dev/vfio/26", O_RDWR);

/* Test the group is viable and available */
ioctl(group, VFIO_GROUP_GET_STATUS, &group_status);

if (!(group_status.flags & VFIO_GROUP_FLAGS_VIABLE))
  /* Group is not viable (ie, not all devices bound for vfio) */

/* Add the group to the container */
ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);

/* Enable the IOMMU model we want */
ioctl(container, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU);

/* Get addition IOMMU info */
ioctl(container, VFIO_IOMMU_GET_INFO, &iommu_info);
```

```
/* Allocate some space and setup a DMA mapping */
dma_map.vaddr = mmap(0, 1024 * 1024,
                     PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
dma_map.size = 1024 * 1024;
dma_map.iova = 0; /* 1MB starting at 0x0 from device view */
dma_map.flags = VFIO_DMA_MAP_FLAG_READ | VFIO_DMA_MAP_FLAG_WRITE;

ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);

/* Get a file descriptor for the device */
device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:06:0d.0");

/* Test and setup the device */
ioctl(device, VFIO_DEVICE_GET_INFO, &device_info);

for (i = 0; i < device_info.num_regions; i++) {
  struct vfio_region_info reg = { .argsz = sizeof(reg) };
  reg.index = i;
  ioctl(device, VFIO_DEVICE_GET_REGION_INFO, &reg);

  /* Setup mappings... read/write offsets, mmaps
   * For PCI devices, config space is a region */
}

for (i = 0; i < device_info.num_irqs; i++) {
  struct vfio_irq_info irq = { .argsz = sizeof(irq) };
  irq.index = i;
  ioctl(device, VFIO_DEVICE_GET_IRQ_INFO, &irq);

  /* Setup IRQs... eventfds, VFIO_DEVICE_SET_IRQS */
}

/* Gratuitous device reset and go... */
ioctl(device, VFIO_DEVICE_RESET);
```


Hide it!

- libvfn reduces this into a **single** API calls
  - `int vfio_pci_open(struct vfio_pci_device *pci, const char *bdf)`
    - Handles group configuration (initializing groups as needed)
    - Lazy container (IOMMU) initialization

What else do we need from a **libvfio**?

- **Container**, **Group** and **Device** initialization helpers ✓
- Other utility functions
  - MMIO helpers ✓
  - portable memory barriers, atomics ✓
  - interrupt configuration ✓
- DMA mapping helpers
  - an I/O Virtual Address allocator ✓
    - libvfn includes a **pretty simple** one (while we wait for the iommufd uAPI)

Address **Translation**

- There are **2** things to solve
  1. What **I/O Virtual Address** to use for a given **Virtual Address**?
     - IOVA equal to VA? IOVAs starting from 0x0?
  2. The ATPTs map **I/O Virtual Addresses** to **Physial Addresses**
     - driver must map **Virtual Addresses** to **I/O Virtual Addresses**
- IOVA **Allocation** and **Lookup** respectively.

IOVA **Allocation**

- **SPDK** does not need an IOVA allocator
  - memory is reserved upfront and pre-mapped
  - **IOVA** is **equal** to **VA**
    - all VAs have predetermined IOVAs

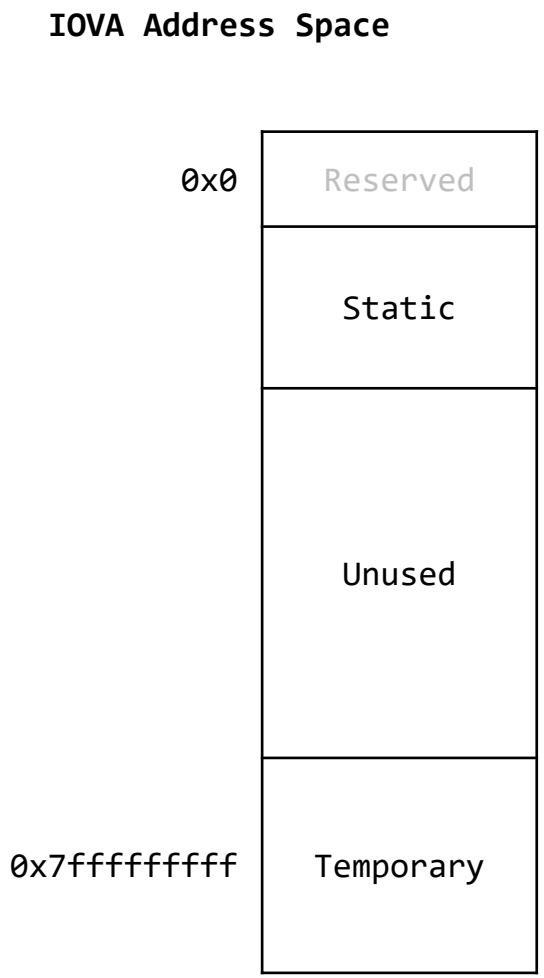
- Applications **must** use SPDKs (DPDKs) memory allocator
  - `spdk_dma_malloc()` and friends
    - not trivial if you want (or **have**) to **Bring Your Own Memory**

IOVA **Allocation**

- **libvfn** does not reserve any memory upfront
  - user must **manually** map any memory to be used with devices
    - sorta Bring Your Own **Mapped** Memory
- Creating a mapping is **expensive** (system call)
  - Some kind of pre-mapping is preferable

IOVA **Allocation**

- libvfn borrows a **simple**, but **effective**, strategy from QEMU
  - **Static** mappings are allocated from **low addresses** going up
    - useful for registered buffers that are reused
    - these are **never** reclaimed
  - **Temporary** mappings are allocated from **high addresses** going down
    - useful for bounce buffering, slow path admin
    - reclaimed when **none in use**
- **We can get rid of this when using iommufd**



IOVA **Lookup**

- While SPDK maps IOVAs equal to VAs, lookup is **still required**
  - to **verify** that a given buffer is mapped
    - a DMA page fault is typically a **catastrophic error**
- SPDK performs a page walk
  - divides the entire 48-bit address space into 2MiB translations
    - 256T → 1G → 2M
  - **Corollary** – Minimum mapping is **2MiB**
    - Fits well with SPDK's use of huge pages

IOVA **Lookup**

- libvfn uses a **skip list** modified for interval lookup
  - Probabilistic data structure with average time complexity comparable to that of a balanced tree
  - Arbitrary (*aligned*) lengths may be mapped

VFIO **Core** Library **API**

libvfn **vfio-core** (iommu api)

- Create and address space container
  - `struct vfio_container *vfio_new(void)`

- Memory **mapping** and **unmapping**
  - `int vfio_map_vaddr(struct vfio_container *vfio, void *vaddr, size_t len, uint64_t *iova);`

  - `int vfio_unmap_vaddr(struct vfio_container *vfio, void *vaddr, size_t *len)`

VFIO **Device** Library **API**

libvfn **vfio-pci**

- **Open** and **initialize** device
  - `int vfio_pci_open(struct vfio_pci_device *pci, const char *bdf)`
- **Configure** IRQs
  - `int vfio_set_irq(struct vfio_device *dev, int *eventfds, int count)`
  - `int vfio_disable_irq(struct vfio_device *dev)`

libvfn **vfio-pci**

- BAR **mapping** and **unmapping**
  - `void *vfio_pci_map_bar(struct vfio_pci_device *pci, int idx, …)`
  - `void vfio_pci_unmap_bar(struct vfio_pci_device *pci, int idx, void *mem, …)`
- **Read** and **write** PCI Configuration Space
  - `ssize_t vfio_pci_read_config(struct vfio_pci_device *pci, void *buf, …)`
  - `ssize_t vfio_pci_write_config(struct vfio_pci_device *pci, void *buf, …)`

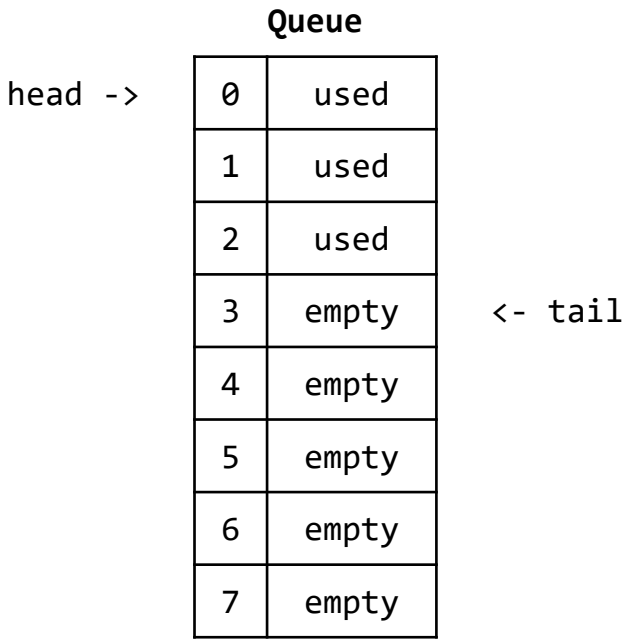
NVMe Library **API**

libvfn **NVMe**

- The “example” NVMe driver in libvfn is about **600** lines of code
  - `src/nvme/core.c`
    - **no** opaque data structures
  - **Minimal** functionality in core
    - bootstraps the admin queue
    - provides helpers to manage I/O submission and completion queues
      - (configure queues for shadow doorbells if supported by controller)

NVMe (super quick) Refresher

- NVMe uses circular lock-free queues for submission and completions
  - **tail** incremented when producing to the queue
  - **head** incremented when consuming from the queue



The **minimal** API for PCIe-based NVMe?

- **4** core queue manipulation functions (the Level **One** API)
  - `void nvme_sq_post(struct nvme_sq *sq, const void *sqe)`
    Copy an SQE to a submission queue
  - `void nvme_sq_update_tail(struct nvme_sq *sq)`
    Notify device about produced SQEs
  - `struct nvme_cqe *nvme_cq_get_cqe(struct nvme_cq *cq)`
    Get pointer to next CQE (might be available, might not)
  - `void nvme_cq_update_head(struct nvme_cq *cq)`
    Notify device about consumed CQEs

The Level **Two** API (nvme_rq)

- May be a **little daunting** to handle commands using the **L1** API
  - Memory for each SQE payload must be manually mapped into the data pointer
    - Requires mapping PRPs, allocating a page for the PRP list if required
  - Each CQE must be matched with the original context of the command

- The **L2** API (`struct nvme_rq`) provides helpers for this
  - Each `struct nvme_sq` is prebaked with a `struct nvme_rq` per SQE
  - Each `struct nvme_rq` is prebaked with memory for PRP list entries

The Level **Two** API (nvme_rq)

- Requests are acquired from and released to SQs as needed
  - `struct nvme_rq *nvme_rq_acquire[_atomic](struct nvme_sq *sq)`
  - `void nvme_rq_release[_atomic](struct nvme_rq *rq)`
    - atomic versions may be useful if a single CQ is associated with multiple SQs and handled in a dedicated thread
  - `void nvme_rq_prep_cmd(struct nvme_rq *rq, union nvme_cmd *cmd)`
- A request may be retrieved from a CQE directly
  - `struct nvme_rq *nvme_rq_from_cqe(struct nvme_ctrl *ctrl, struct nvme_cqe *cqe)`

The Level **Two** API (nvme_rq)

- A **contiguous** buffer may be mapped into the PRPs
  - `int nvme_rq_map_prp(struct nvme_rq *rq, union nvme_cmd *cmd, uint64_t iova, size_t len)`
- Or a **discontigous** buffer (as described by a struct iovec)
  - `int nvme_rq_mapv_prp(struct nvme_rq *rq, union nvme_cmd *cmd, struct iovec *iov, int niov)`

Examples

Basic Example (read register)

```
struct nvme_ctrl ctrl = {};

/* configure and enable controller (default options) */
nvme_init(&ctrl, “0000:01:00.0”, NULL)

void *regs = ctrl.regs;

/* read BAR0 (the NVME MBAR) register */
uint64_t cap = le64_to_cpu(mmio_read64(regs + NVME_REG_CAP));

/* print a value from the register */
printf(“CAP.MQES %lx\n”, NVME_CAP_MQES(cap));
```


Identify Example (issue command asynchronously)

```
/* allocate and map memory */
size_t len = pgmap(&vaddr, NVME_IDENTIFY_DATA_SIZE);
vfio_map_vaddr(vfio, vaddr, NVME_IDENTIFY_DATA_SIZE, &iova);

/* setup command */
cmd.identify = (struct nvme_cmd_identify){.opcode = nvme_admin_identify, .cns = NVME_IDENTIFY_CNS_CTRL};

struct nvme_rq *rq = nvme_rq_acquire(ctrl.adminq.sq);

nvme_rq_map_prp(rq, &cmd, iova, NVME_IDENTIFY_DATA_SIZE);

nvme_rq_exec(rq, &cmd); /* post and ring doorbell */
nvme_rq_spin(rq, &cqe); /* spin on cq */

nvme_rq_release(rq);

printf("vid 0x%"PRIx8"\n", (struct nvme_id_ctrl *)vaddr->vid);
```


Identify Example (eventfd)

```
int efd = eventfd(0, 0);

/* register eventfd for vector 0 */
vfio_set_irq(&ctrl.pci.dev, &efd, 1);

nvme_rq_map_prp(rq, &cmd, iova, NVME_IDENTIFY_DATA_SIZE);
nvme_rq_exec(rq, &cmd);

/* wait for interrupt */
uint64_t v;
read(efd, &v, sizeof(v));

/* will not spin */
nvme_rq_spin(rq, &cqe);
```


Case **Study**

Integration with xNVMe

Integration in **xNVMe**

- **libvfn** is available as an **alternative** to the SPDK backend in xNVMe
  - makes xNVMe a little lighter
- xNVMe requires backends to implement
  - buffer **allocation** and **mapping**
    - maps directly to mmap (allocation) and DMA mapping of the buffers
  - async interface
    - queue init, poke, (vectored) io

Integration in **xNVMe**

- The asynchronous interface in xNVMe is based on callbacks
  - libvfn's `struct nvme_rq` opaque member stores the xNVMe command context
    - holds the callback and argument to be executed upon command completion
- The xNVMe asynchronous API maps almost 1-to-1 with libvfn
  - queue init
    - `nvme_create_ioqpair()`
  - io
    - `nvme_rq_post/exec()`
  - poke
    - loop around `nvme_cq_get_cqe()` and `nvme_rq_from_cqe()`

Performance Numbers

Are we **on par**?

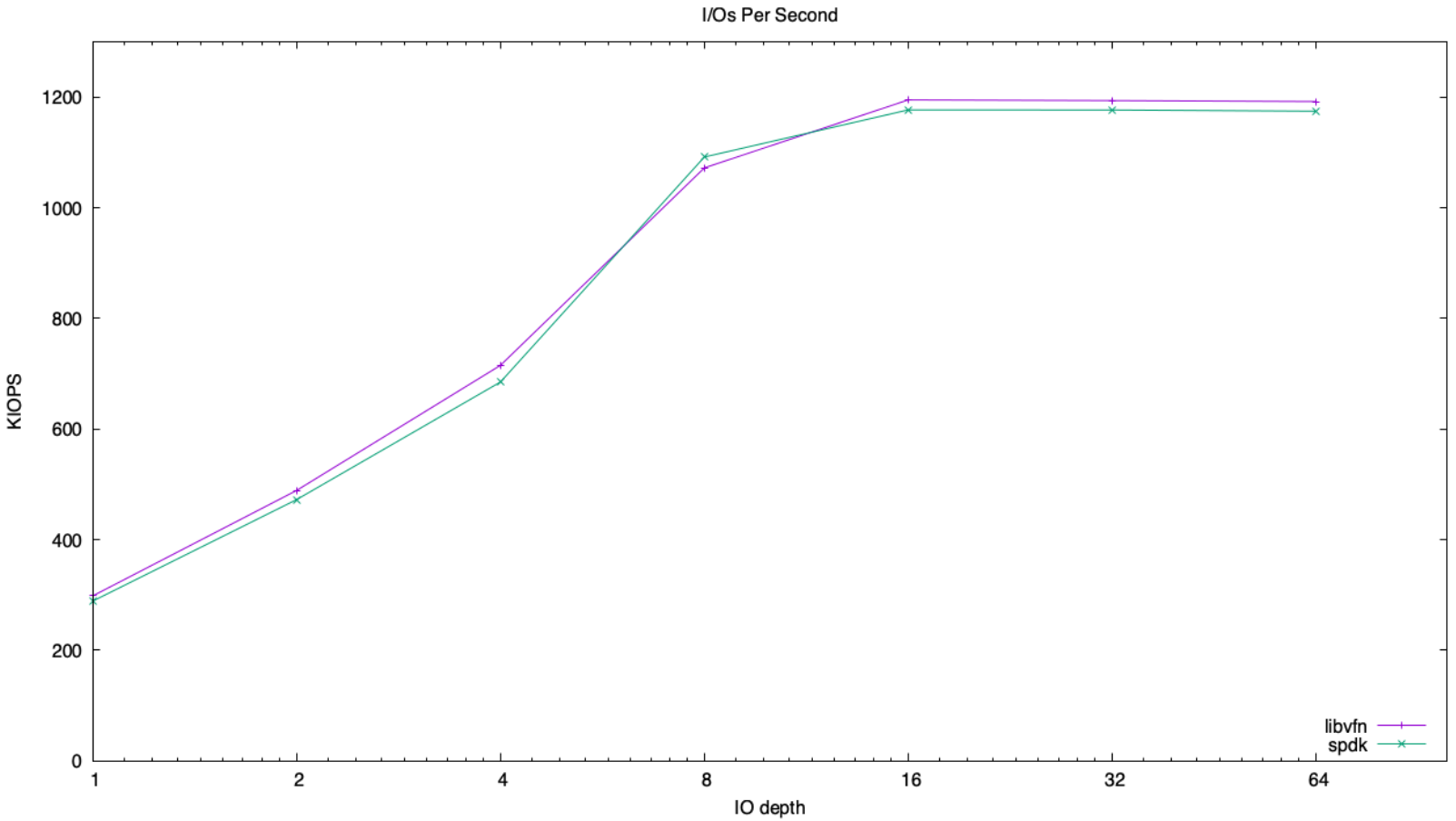
Performance (**Setup**)

- Intel(R) Xeon(R) CPU E3-1240 v6 @ 3.70GHz
  - An oldie, but a goodie…
- Intel NVMe Optane Memory Series
  - MEMPEK1W016GA
    - 16GB, M.2 80mm PCIe 3.0, 20nm, 3D Xpoint™
- **1** core (**1** thread, **1** queue) – **random read (512 bytes)**
  - NVMe queue size is **128** (device max)
  - I/O queue depths **1**, **2**, **4**, **8** … **64**
  - **10s** warmup, **30s** proper

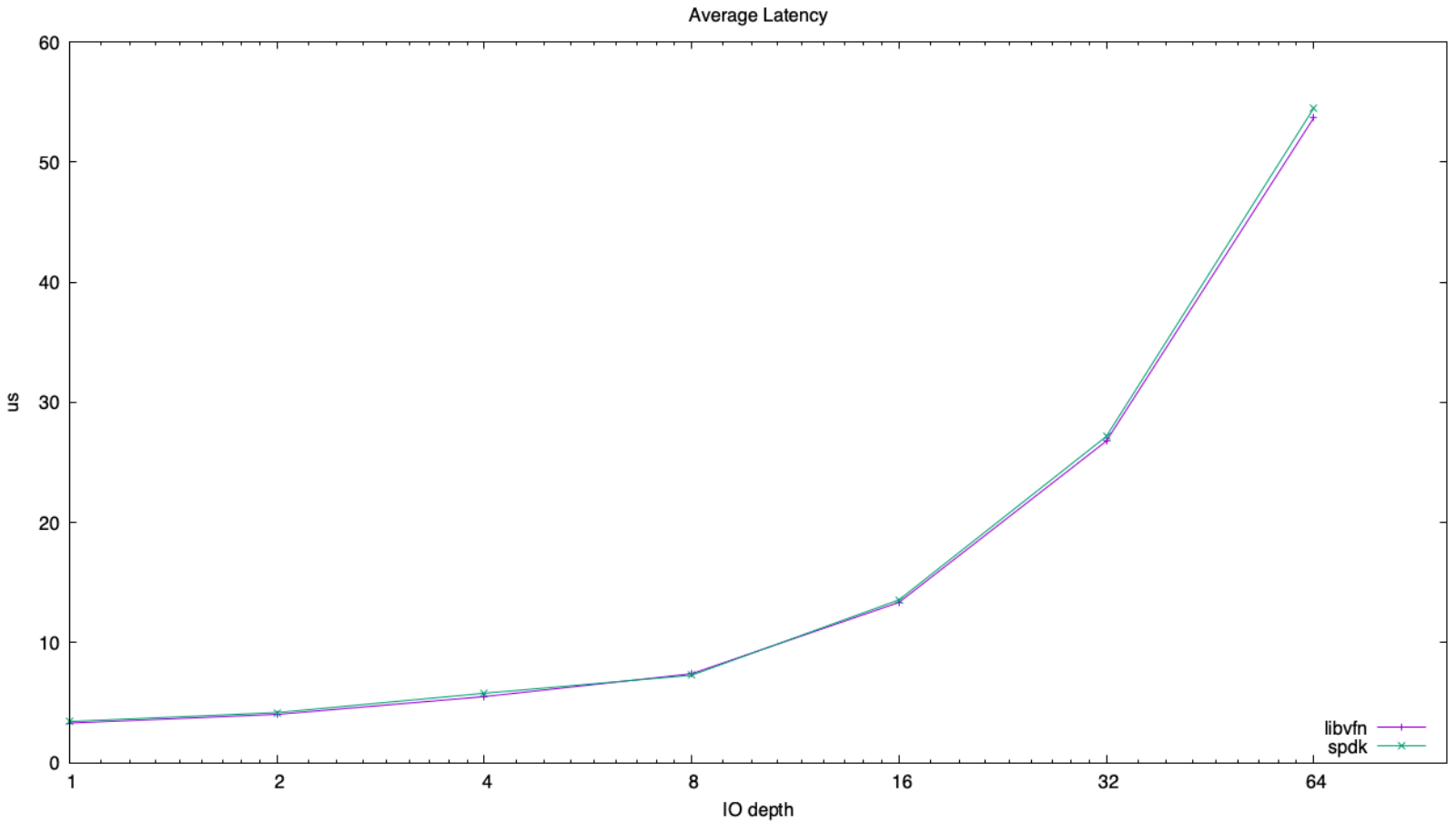
Performance (**program**)

- As close to an **apples-to-apples** comparison with SPDK as possible
  - libvfn `examples/perf.c` mirrors spdk `examples/nvme/perf/perf.c`
    - Re-issue command on completion
    - Same time measurement strategy (RDTSC-based, not `clock_gettime`)
    - Everything pre-allocated

Performance (**IOPS**)



Performance (**Average Latency**)



Wrapping Up

Examples and Next Steps

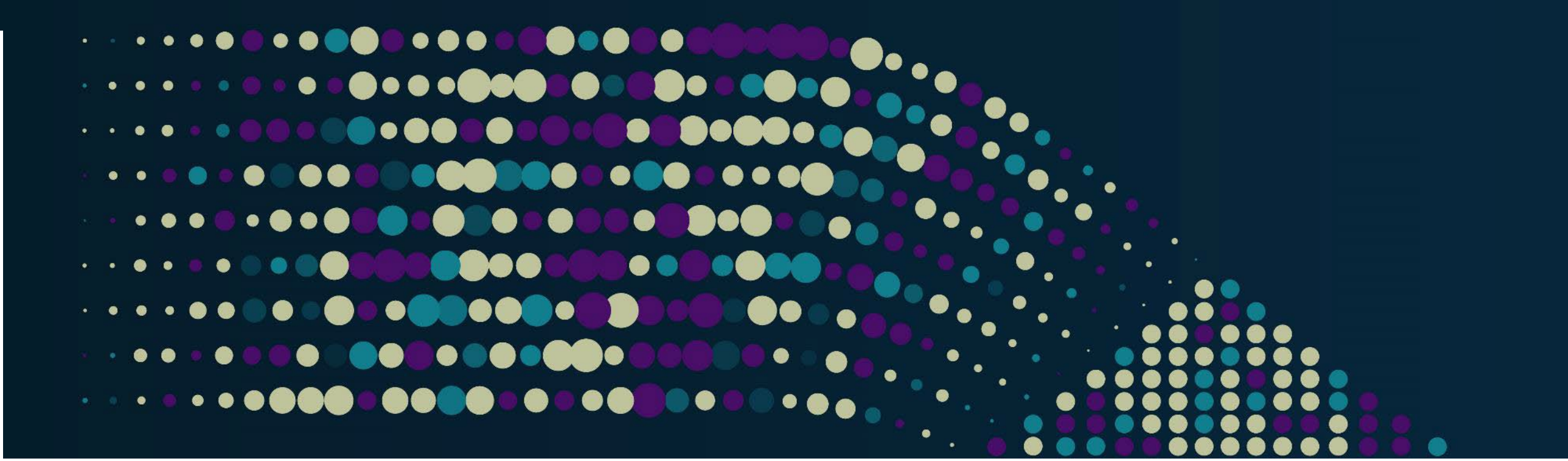
What about **iommufd**?

- **Yes**.

- We are following the latest developments and **testing**
  - My colleague, **Joel Granados**, is integrating iommufd support
  - Rework vfio parts to use this as appropriate
  - Does requires some public API changes (planned for **v4**)
    - `vfio_{map,unmap}_vaddr()` → `iommu_{map,unmap}_vaddr()`
    - We already hide the group-centric VFIO API behind a device centric abstraction

Next Steps

- More NVMe **helpers**
  - SGL mapping helpers
  - Pluggable IOVA allocation and lookup
  - More sugar? Maybe in another support library?
- **Non 4KB** page size based systems
  - Some assumptions in the core. **Mads** is working on that.
- **WONTFIX'es**
  - High level event framework (roll your own)



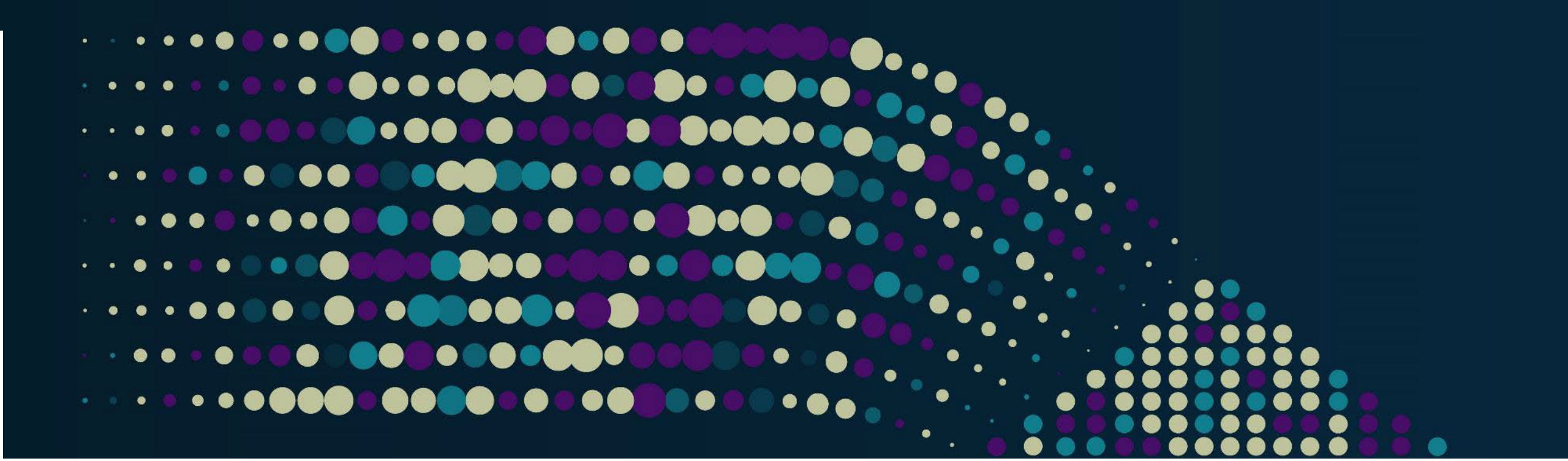
Questions?

Grab libvfn at **github.com/OpenMPDK/libvfn**

stable **v2**

v3 just released

SDC 23



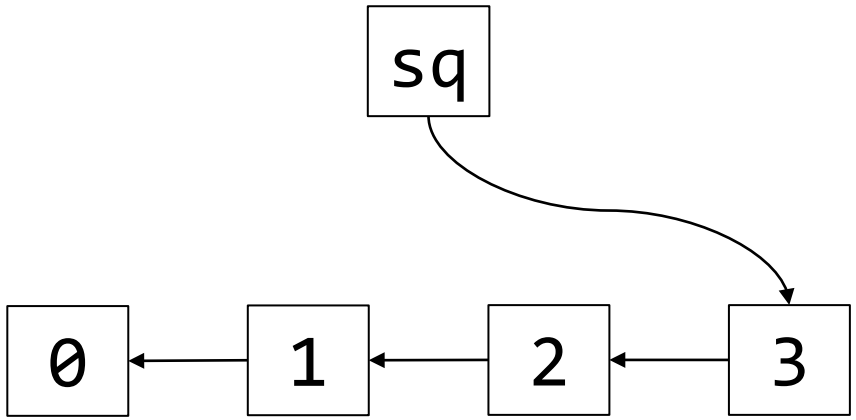
Please take a moment to rate this session.

Your feedback is important to us.

Backup for Questions

The Level **Two** API (nvme_rq)

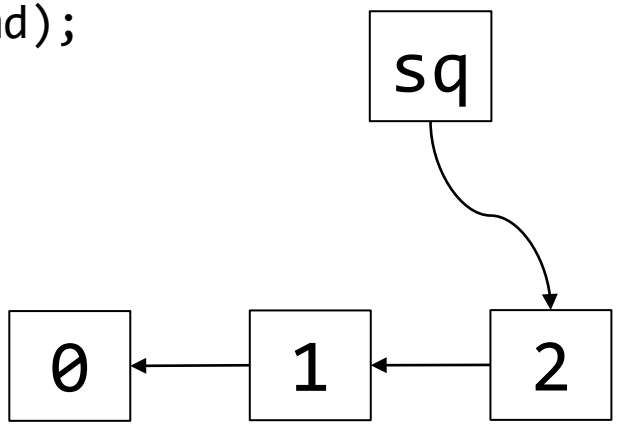
executing

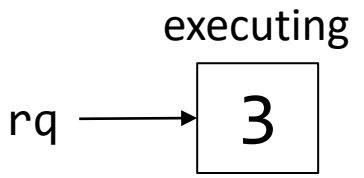


completed

The Level **Two** API (nvme_rq)

```
rq = nvme_rq_acquire(sq);
nvme_rq_exec(rq, cmd);
```

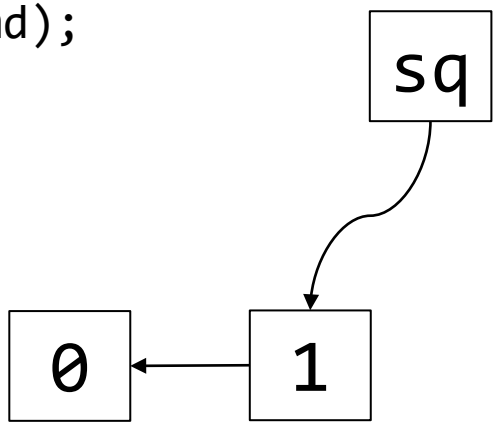


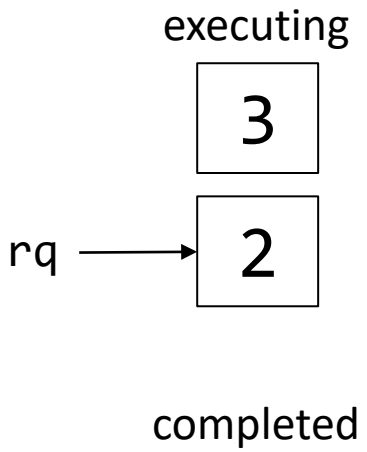


completed

The Level **Two** API (nvme_rq)

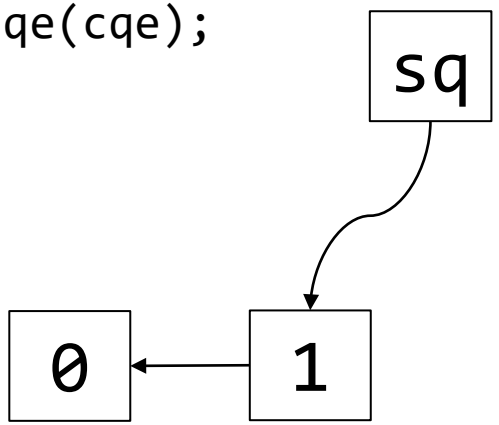
```
rq = nvme_rq_acquire(sq);
nvme_rq_exec(rq, cmd);
```

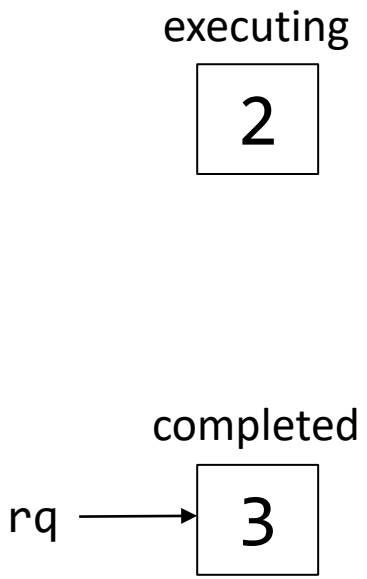




The Level **Two** API (nvme_rq)

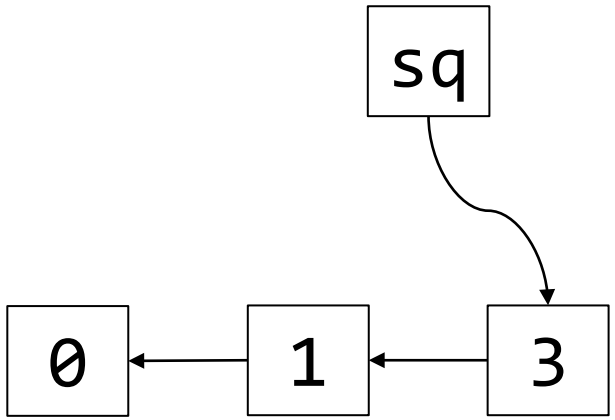
```
cqe = nvme_cq_get_cqe(cq);
rq = nvme_rq_from_cqe(cqe);
```





The Level **Two** API (nvme_rq)

```
nvme_rq_release(rq);
```



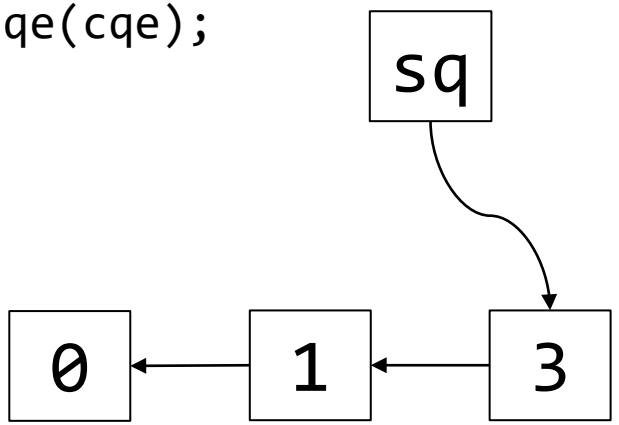
executing

2

completed

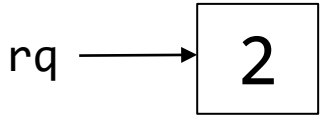
The Level **Two** API (nvme_rq)

```
cqe = nvme_cq_get_cqe(cq);
rq = nvme_rq_from_cqe(cqe);
```



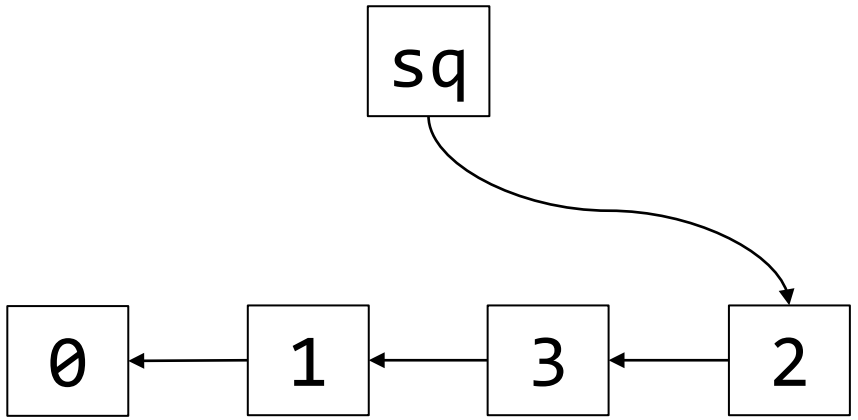
executing

completed



The Level **Two** API (nvme_rq)

```
nvme_rq_release(rq);
```



executing

completed

nvme_rq_acquire()

Regular

```
struct nvme_rq *rq = sq->rq_top;
if (!rq) {
  errno = EBUSY;
  return NULL;
}

sq->rq_top = rq->rq_next;

return rq;
```

Atomic

```
struct nvme_rq *rq = load_acquire(&sq->rq_top);

while (rq && !cmpxchg(&sq->rq_top, rq, rq->rq_next))
  ;

if (!rq)
  errno = EBUSY;

return rq;
```


nvme_rq_release()

Regular

```
struct nvme_sq *sq = rq->sq;

nvme_rq_reset(rq);

rq->rq_next = sq->rq_top;
sq->rq_top = rq;
```

Atomic

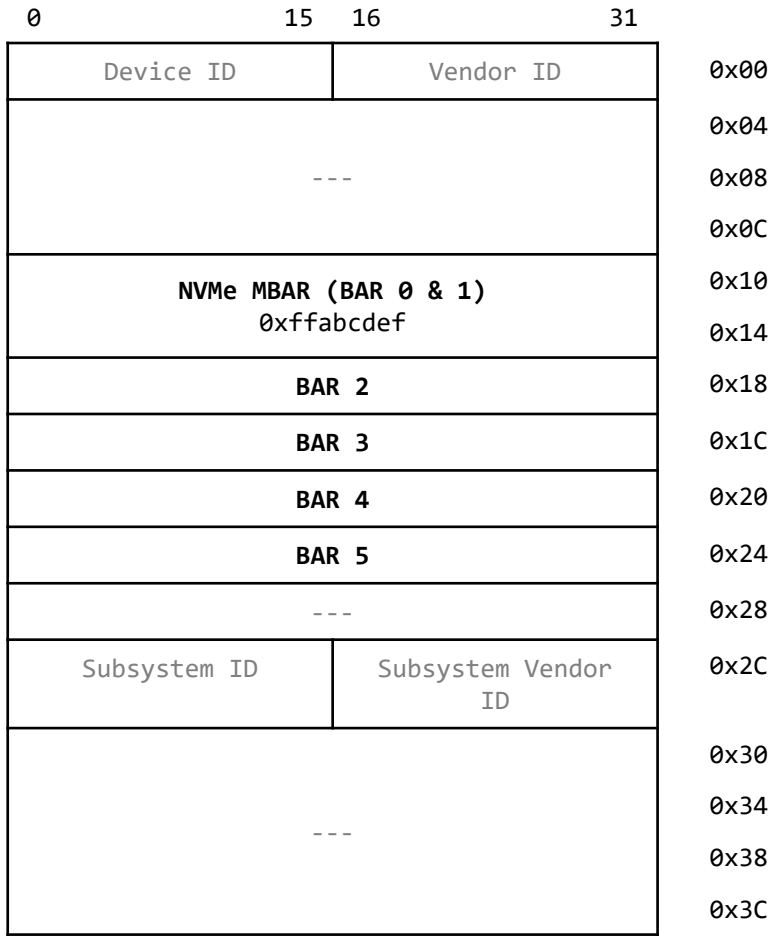
```
struct nvme_sq *sq = rq->sq;

nvme_rq_reset(rq);

rq->rq_next = load_acquire(&sq->rq_top);

while (!cmpxchg(&sq->rq_top, rq->rq_next, rq))
  ;
```


NVMe **Refresher**

… it’s just queue processing

NVMe **Refresher**

- Host informs the device about new entries in the queue using a “doorbell” mechanism
  - A “doorbell” is the common name for a write-only memory-mapped I/O register
- PCI devices expose these registers in the PCI Configuration Space
  - In NVMe, the controller registers are located in the NVMe “MBAR” (BAR 0 & 1)



NVMe **Refresher**

Start	End	Size	Symbol	Description
0x0	0x07	8	**CAP**	Controller Capabilities
0x08	0x0b	4	**VS**	Version
0x14	0x17	4	**CC**	Controller Configuration
0x28	0x2f	8	**ASQ**	Admin Submission Queue Base Address
0x30	0x37	8	**ACQ**	Admin Completion Queue Base Address
...				
0x1000	0x1003	4	**SQ0TBDL**	Submission Queue 0 Tail Doorbell (Admin)
0x1004	0x1007	4	**CQ0HDBL**	Completion Queue 0 Head Doorbell (Admin)
0x1008	0x100b	4	**SQ1TBDL**	Submission Queue 1 Tail Doorbell
0x100c	0x100f	4	**CQ1HBDL**	Completion Queue 1 Head Doorbell
0x1000 + (2n << 2)	0x1003 + (2n << 2)	4	**SQnTBDL**	Submission Queue n Tail Doorbell
0x1000 + (2n+1 << 2)	0x1003 + (2n+1 << 2)	4	**CQnHDBL**	Completion Queue n Head Doorbell

NVMe **Refresher**

Start	End	Size	Symbol	Description
0x0	0x07	8	**CAP**	Controller
0x08	0x0b	4	**VS**	Version
0x14	0x17	4	**CC**	Controller
0x28	0x2f	8	**ASQ**	Admin Subm
0x30	0x37	8	**ACQ**	Admin Completion Queue Base Address
...				
0x1000	0x1003	4	**SQ0TBDL**	Submission Queue 0 Tail Doorbell (Admin)
0x1004	0x1007	4	**CQ0HDBL**	Completion Queue 0 Head Doorbell (Admin)
0x1008	0x100b	4	**SQ1TBDL**	Submission Queue 1 Tail Doorbell
0x100c	0x100f	4	**CQ1HBDL**	Completion Queue 1 Head Doorbell
0x1000 + (2n << 2)	0x1003 + (2n << 2)	4	**SQnTBDL**	Submission Queue n Tail Doorbell
0x1000 + (2n+1 << 2)	0x1003 + (2n+1 << 2)	4	**CQnHDBL**	Completion Queue n Head Doorbell

Using **MMIO**, the host writes the tail/head values to the relevant doorbells

“Ringing the Doorbell”

