SEE TEXT ONLY

FreeBSD and NVM EXPRESS

By Jim **Harris** and Warner **Losh**

NVM Express (NVMe) has quickly become the predominant standard for high-performance, non-volatile memory access across PCI Express. FreeBSD added support for NVMe in 2012, enabling FreeBSD to take advantage of devices that can deliver over 500,000 IO/s per NVMe device[1]. Companies like Netflix have rapidly moved to deploying NVMe storage on FreeBSD, such that FreeBSD's NVMe subsystem is now helping drive a significant portion of North American internet traffic[2]. In this article, we describe the NVMe specification and FreeBSD's implementation of the specification and provide an overview of FreeBSD's utilities for monitoring and managing NVMe storage.

Let's first describe some of the common terms used in NVMe. An NVMe SSD is referred to as an NVMe controller and is roughly equivalent to a SCSI HBA or an AHCI controller. The primary difference is that for NVMe, the media resides within the controller itself – there is no separate protocol nor cabling between the controller and its media. The storage media within the NVMe controller is grouped into one or more NVMe namespaces. A namespace can be considered roughly equivalent to a SCSI LUN. This combination of the HBA and media into one unit reduces the amount of overhead by simplifying the protocol needed and eliminating layers of abstraction. The design uses the capabilities of PCIe to eliminate driver bottlenecks by supporting lockless queueing of requests.

In FreeBSD, NVMe controllers and namespaces are enumerated and initialized through the `nvme(4)` driver. `nvme(4)` is also responsible for providing an interface to expose these namespaces as block devices, but does not register those namespaces with GEOM nor CAM directly. `nvd(4)` registers each namespace with GEOM as a block device. Since NVMe has become mainstream since `nvd(4)` was originally developed, Netflix added `nda(4)` as an alternative to `nvd(4)`. The `nvd(4)` driver is a very thin layer on top of the NVMe protocol and is designed to operate at high transaction rates. The `nda(4)` driver integrates with CAM, including its error recovery and advanced queueing. It also offers traffic shaping to the drive via the I/O scheduler to improve overall performance.

NVMe System Integration

As you can see from this idealized picture, `nvd(4)` integrates directly to the disk layer. This has the advantage of queueing the commands to the NVMe drive with as little delay as possible. Since NVMe drives are designed to scale, this works out fairly well. The minimal delay in getting I/O to the drive translates to a lower latency and simpler code path.

In contrast, the `nda(4)` driver connects through CAM. CAM schedules the I/O to the

<table>
<tr><td colspan="2">User Space</td></tr>
<tr><td colspan="2">System Calls</td></tr>
<tr><td colspan="2">File System</td></tr>
<tr><td colspan="2">Buffer Cache</td></tr>
<tr><td colspan="2">GEOM</td></tr>
<tr><td colspan="2">Disk Interface</td></tr>
<tr><td></td><td>nda</td></tr>
<tr><td>nvd</td><td>CAM</td></tr>
<tr><td></td><td>nvme_sim</td></tr>
</table>

drive. This can involve reordering commands at times. The CAM system is designed to accept a high queue rate as to have the queued I/O sent to the drive as the I/O is available for it. CAM also adds error handling which can help recover drives when things go wrong better than `nvd(4)` (which has little error recovery in its error path). `nda(4)` can also shape I/O traffic to the drive, to a limited extent, through the CAM I/O scheduler. This can be utilized to introduce some unfairness to produce better results. For example, video streaming benefits from a bias towards read operations. Many NVMe drives have trouble with TRIMs but will soon be able to heavily throttle or shape the trims to the drive. `nda(4)` is able to collapse multiple trims into one trim request for the drive.

The `nvme(4)` driver is responsible for enumerating the nvme drives and their namespaces when the driver is attached. The `nvd(4)` or `nda(4)` drivers register interest in these (and other) events with the `nvme_register_consumer()` call at startup. So, when it detects a new namespace, it calls the new namespace callback. For the `nvd(4)` driver, a new disk interface is created and registered with GEOM. With the `nda(4)` driver, the callback goes to `nvme_sim` which creates the appropriate CAM devices that cause `nda(4)` to be created (CAM's SCSI legacy means that it does many things via indirection or deferred callback). The `nda(4)` driver then creates the disk and registers it with GEOM.

Regardless of the disk interface, when requests come from the upper layers of the system (specifically from the disk layer via command structures called bios), the `nvd` or `nda` will convert the `BIO_READ, BIO_WRITE` and `BIO_DELETE` commands into the appropriate `nvme` command and pass those requests to the `nvme(4)` driver for execution.

Exposing NVMe namespaces via `nvd(4)` versus `nda(4)` is controlled by the `hw.nvme.use_nvd` tunable. It currently defaults to 1, meaning namespaces are exposed via `nvd(4)`.

NVMe Commands and Queue Pairs

NVMe commands such as `READ`, `WRITE` and `IDENTIFY` are submitted by the host to the controller via submission queues, with the controller notifying the host about completions of those commands via completion queues. The specification allows for multiple submission queues to share one completion queue, but FreeBSD always associates one submission queue with one completion queue to form a logical queue pair or "qpair". These qpair associations are critical for unlocking NVMe parallelism on many-core systems and will be described later in this article.

Submission queues are a contiguous host memory region acting as a circular buffer. Each submission queue entry is 64 bytes. The host notifies the controller of new commands by filling out the next entry in the queue and then writing to a per-queue doorbell in the controller's MMIO register space. In `nvme(4)`, this is done in `nvme_qpair_submit_tracker()`:

```
696     /* Copy the command from the tracker to the submission queue. */
697     memcpy(&qpair->cmd[qpair->sq_tail], &req->cmd, sizeof(req->cmd));
698
699     if (++qpair->sq_tail == qpair->num_entries)
700             qpair->sq_tail = 0;
701
702     wmb();
703     nvme_mmio_write_4(qpair->ctrlr, doorbell[qpair->id].sq_tdbl,
704         qpair->sq_tail);
```

Completion queues are a separate contiguous host memory region with 16-byte entries. The controller notifies the host about completions by filling out the next entry in the completion queue. This entry contains both the submission queue's ID and the per-queue command ID that the host used when submitting the command. The controller then interrupts the host and the driver can start processing completion entries. NVMe defines a phase bit in each completion queue entry to enable the host to determine which completion entries are valid. The controller will write this phase bit to 1 the first time through the queue, and then alternating between `0` and `1` through subsequent passes through the queue. The driver checks each completion queue entry's phase bit against the expected value to determine which entries contain new completions. In `nvme(4)`, this is done in `nvme_qpair_process_completions()`:

```
424       while (1) {
425               cpl = &qpair->cpl[qpair->cq_head];
426
427               if (cpl->status.p != qpair->phase)
428                       break;
429
```

This method of completion queue processing enables the host to only read memory to determine which commands have been completed. This ensures there are no MMIO reads in the performance code path. It also takes advantage of CPU features such as Intel®'s Data Direct I/O Technology[3] to place completion entries directly in last-level cache to optimize completion entry processing.

NVMe commands are further divided into two types – admin and I/O – which are handled by two different types of qpairs – also named admin and I/O. The queue processing described earlier applies identically to both admin and I/O qpairs, but the method of initializing them is quite different. Walking through how an NVMe controller is initialized will help with this understanding.

NVMe Controller Initialization

At a high level, controller initialization is split into two stages:

Controller reset – this stage is performed by the host using NVMe MMIO register reads and writes. Its primary functions are describing the parameters of the admin qpair (host memory addresses and queue sizes) and then toggling an enable bit (`CC.EN`). After the enable bit is toggled, the controller will start its internal initialization and set its ready bit (`CSTS.RDY`) when it is completed. The host waits for the ready bit to become set before proceeding to the next stage.

Controller setup – this stage is performed by the host using admin commands which can now be submitted on the admin qpair that was setup in the previous stage.

- Submit an `IDENTIFY` command with command dword 10 (`CDW10`) set to 1. This signifies the controller to return the `IDENTIFY` data associated with the controller.
- Submit `CREATE_IO_CQ` and `CREATE_IO_SQ` commands to construct I/O qpairs. The number of I/O qpairs allocated typically equals the minimum of the number of CPU cores on the system and the maximum number of I/O qpairs supported by the controller. This will be described in more detail later in this article.
- Submit an `IDENTIFY` command for each namespace reported by the controller's `IDENTIFY` data. The key piece of data in the namespace's `IDENTIFY` data is the size and format of the namespace. The namespace is specified in the `IDENTIFY` command using the NSID field which always starts at 1 (there is never a namespace 0)!

I/O Queue Allocation

I/O qpair allocation is a key feature of the `nvme(4)` driver. With many I/O qpairs at our disposal, we can ideally allocate a separate qpair per CPU core. This enables threads on each CPU core to submit I/O commands without synchronization with threads running on other CPU cores. It also enables binding completion interrupts to the CPU core where the I/O was submitted to improve cache locality.

Practically, there are a couple of reasons why a per-core qpair may not be possible. First, while the NVMe specifications allow for up to 65,535 I/O qpairs per controller, most NVMe SSDs allow far fewer I/O qpairs – sometimes fewer than 32. Second, there may be a limited number of interrupt vectors available on the system. This latter limitation was exposed most frequently on systems with many NVMe SSDs and multi-queue NICs but should now be much less likely to occur thanks to some SMP improvements that are now in FreeBSD 11.[4]

For cases where a qpair cannot be allocated per CPU core, `nvme(4)` will allocate as many qpairs as it can, and then associate each qpair with multiple CPU cores. A mutex is used to synchronize access to the qpair – not only between different threads submitting I/O, but also with the qpair's completion handler.

I/O Submission

An NVMe command primarily consists of the following and is represented by FreeBSD's struct `nvme_command` found in `/usr/include/dev/nvme/nvme.h`:

- An 8-bit opcode – admin and I/O opcodes overlap but are easily differentiated based on its submission queue type
- A 16-bit command ID – this command ID must be unique among any other commands submitted on the same queue
- Namespace identifier – primarily used for I/O commands, but some admin commands such as `IDENTIFY` also use this field; note that this infers any I/O queue and submit I/O to any namespace
- Two Physical Region Page (PRP) entries

PRP is NVMe's version of scatter-gather lists (SGL). Instead of scatter-gather elements which typically specify a start address and length, PRP entries only specify a start address. The length is inferred by the distance of the start address from the next page, and the overall size of the I/O command. It also has limitations such as all PRP entries except the first must start on a 4KiB boundary, and all entries except the first and last must describe exactly a 4KiB length buffer. Fortunately, these limitations do not affect FreeBSD, since the FreeBSD I/O stack always uses virtually contiguous buffers which can always be represented by PRP.

An astute reader will notice that an NVMe command only has two PRP entries – so how do we represent the I/O buffers for large commands? In these cases, the second PRP entry points to a list of other PRP entries. But this means that for large I/O, we need buffers for the PRP lists that are mapped for DMA.

`nvme_request` *and* `nvme_tracker`

There are two considerations for PRP lists:

- PRP list buffers need to be mapped for DMA, so PRP list buffers are allocated and mapped when the qpair is allocated. This avoids mapping the PRP list in the performance path.
- PRP list buffers can become large in certain environments. The size of a PRP list buffer is proportional to MAXPHYS. FreeBSD's default MAXPHYS is 128KiB which results in a maximum PRP list of 256 bytes. Netflix however deploys with MAXPHYS of 1MiB which results in 2KiB PRP list buffer size. So, we need to be judi cious in how many PRP lists are allocated per qpair. The `nvme` driver automatically limits the request size to the smaller of MAXPHYS and 2MiB, the largest buffer that fits in the two PRPs pages available for the SG list.

`nvme(4)` defines two different structures – `struct nvme_request` and `struct nvme_tracker`. When an I/O qpair is allocated, 128^5 `nvme_trackers` are allocated, along with a PRP list buffer for each `nvme_tracker`. This represents the maximum number of I/O that can be outstanding on the qpair at any given time.

struct nvme_request represents a command that has been requested by the caller. The simplest case is `nvme_ns_cmd_read()`:

```
32 int
33 nvme_ns_cmd_read(struct nvme_namespace *ns, void *payload, uint64_t lba,
34     uint32_t lba_count, nvme_cb_fn_t cb_fn, void *cb_arg)
35 {
36         struct nvme_request   *req;
37
38         req = nvme_allocate_request_vaddr(payload,
39           lba_count*nvme_ns_get_sector_size(ns), cb_fn, cb_arg);
40
41         if (req == NULL)
42                 return (ENOMEM);
43
44         nvme_ns_read_cmd(&req->cmd, ns->id, lba, lba_count);
45
46         nvme_ctrlr_submit_io_request(ns->ctrlr, req);
47
48         return (0);
49 }
```

Here the caller is requesting to read data from the namespace into the buffer "payload". First, `nvme_allocate_request_vaddr()` allocates an `nvme_request structure` with the parameters specified by the caller. Next, the `nvme_command structure` is populated by `nvme_ns_read_cmd()`. Note that `nvme_command` is a 64-byte submission queue entry, but at this point we are only preparing the

submission queue entry – it will get copied later into the actual submission queue. Finally, we call `nvme_ctrlr_submit_io_request()`.

```
1207  void
1208  nvme_ctrlr_submit_io_request(struct nvme_controller *ctrlr,
1209      struct nvme_request *req)
1210  {
1211          struct nvme_qpair       *qpair;
1212
1213          qpair = &ctrlr->ioq[curcpu / ctrlr->num_cpus_per_ioq];
1214          nvme_qpair_submit_request(qpair, req);
1215  }
```

Here is where we pick the qpair based on the current CPU. Now that we know which qpair to use, we call `nvme_qpair_submit_request()`. This function checks if there are any available `nvme_trackers`. If there are, it calls `nvme_qpair_submit_tracker()` which we saw earlier. If not, it puts the `nvme_request` in an STAILQ. Later, once some I/O are completed, this STAILQ will be checked and `nvme_requests` are resubmitted.

NVMe Namespaces

The NVMe specification allows the logical partitioning of drives into individual namespaces. A namespace is nothing more than a collection of blocks, addressed 0..N-1. These namespaces may be fully provisioned or thinly provisioned (and in fact, hardware or hypervisors that emulate NVMe drives often hide these details from the host). Namespaces also provide ways to independently assign attributes to those namespaces. One namespace may be used to store the OS. The data rarely changes, usually has few writes, but a lot of reads. Another namespace may contain a transaction log which is read infrequently but written in a mostly append I/O pattern. Still another may contain data that's rewritten all the time and very hot. The firmware on the NVMe drive can use these attributes to optimize NAND storage. For cold data, like the OS, the firmware may place it into cells that have low wear and are storing 3 bits per cell to maximize data density. These cells often have excellent long-term retention. For very hot data, the drive may choose to use cells that are more worn and may store a lot of it in 1-bit-per-cell to maximize speed. While the worn cells cannot retain data as long, they are idea for hot data because the data won't be stored for long anyway, so any deficiency in longevity will not hinder the drive's performance.

FreeBSD does not yet support namespace management – for example, creating and deleting namespaces and specifying attributes for those namespaces. There is active work in this area within the FreeBSD community however and should make it in time for FreeBSD 12.

Managing NVMe Drives

The primary utility for listing and configuring NVMe controllers and namespaces is `nvmecontrol(8)`. The most basic `nvmecontrol` subcommand is "`nvmecontrol devlist`" which provides a brief summary of the each NVMe controller and its `namespace(s)`.

```
%sudo nvmecontrol devlist
nvme0: ORCL-VBOX-NVME-VER12
 nvme0ns1 (1024MB)
```

Additional `nvmecontrol(8)` subcommands include:

- "`nvmecontrol identify`" for providing details on NVMe controllers and namespaces based on information from the `NVMe IDENTIFY` command
- "`nvmecontrol logpage`" for reading log pages from an NVMe controller; specification-defined log pages for Errors, Health/SMART, and Firmware Slots have handlers to translate the log page into a human-readable format
- "`nvmecontrol firmware`" for downloading and/or activating different firmware images on an NVMe controller
- "`nvmecontrol perftest`" for running a low-level performance test from the `nvme(4)` driver itself

- "`nvmecontrol reset`" to issue a controller-level reset to an NVMe controller
- "`nvmecontrol power`" to change the power state or specify a workload hint for an NVMe controller
- "`nvmecontrol wdc`" to perform options specific to WDC NVMe SSDs

`camcontrol(8)` can currently be used to list namespaces when `nda(4)` is in use, but other functionality such as NVMe identify or firmware download are not yet plumbed.

One shortcoming of `nvmecontrol(8)` is mapping an NVMe controller or namespace to its associated `nvd` or `nda` entry. The best way to do this currently is associating serial numbers between "`geom disk list`" and "`nvmecontrol identify`".As `camcontrol(8)` gains more NVMe functionality, this issue will be mitigated.

Summary

NVM Express provides a modern, high performance storage interface well suited to today's CPU architectures – and FreeBSD is well-positioned to take advantage of it. Integrating NVMe support with CAM, as well as future support for namespace management and NVMe SSD hotplug will further improve on FreeBSD's NVMe capabilities. Next time you watch Netflix, you will hopefully know a little bit more about how those bits ended up on your screen! ●

[1] https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-900p-brief.html
[2] https://www.sandvine.com/hubfs/downloads/archive/2016-global-internet-phenomena-report-latin-america-and-north-america.pdf
[3] https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html
[4] https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=199321
[5] 128 is the default. This number can be modified with the hw.nvme.io_trackers tunable.

Jim Harris is a Principal Software Engineer in Intel's Data Center Group and was granted his FreeBSD source commit bit in 2011. He is the original author of the FreeBSD nvme and nvd drivers as well as the nvmecontrol management utility. Jim has also helped bring libraries and tools such as DPDK and Intel VTune Amplifier to FreeBSD and ported the FreeBSD nvme driver to userspace as part of his current role as Storage Performance Development Kit software architect.

Warner Losh is a Senior Software Engineer at Netflix, where he optimizes FreeBSD's storage system for video delivery servers. He has been a FreeBSD contributor for over 20 years and is currently serving his sixth term on the FreeBSD core team. Warner has improved a number of systems—most recently the boot loader—in FreeBSD. Prior to Netflix, he produced flash drives and measured atomic clocks for accuracy. His code still measures some of the clocks that create UTC!