
MSN：并行数据存储研究组		产品版本	V1.0	密级	
项目名称		项目 ID		共 页	

SPDK 简介

Version 1.0

编写	甘泉	日期	2021/1/27
审核		日期	
批准		日期	

修 订 记 录

日期	版本	描述	作者
2021/1/27	V1.0	介绍初版	甘泉

目录

SPDK 介绍与使用	4
1 背景	4
1.2 系统 I/O 介绍	4
1.2 用户态驱动	5
2、SPDK 介绍	8
2.1 APP Scheduling	8
2.2 Drivers	11
2.3 Storage services	12
2.4 Storage Protocol	15
3 SPDK 的配置和性能评估	17
3.1 SPDK 配置	17
3.2 SPDK 的性能评估	18
4 应用场景及其不足	20
5 实例分析	21

SPDK 介绍与使用

1 背景

1.2 系统 I/O 介绍

NVMe 标准的将存储产品从 AHCI（串行 ATA 高级主机控制器接口）中解放出来，使得固态硬盘产品能够发挥出更高的性能优势。系统 I/O 性能的影响主要在于固态硬盘的硬件性能和软件的开销。从持续满足上层应用的高性能的角度看，有两种途径：一是开发更高性能的固态硬盘硬件设备；二是减少软件的开销。如图 1-1，基于最新 3D XPoint 技术的 Intel Optane NVMe SSD 设备可以在延迟和吞吐量方面使得性能更上一层楼，这也导致大量的软件开销，包括，上下文切换、数据拷贝、频繁的中断以及线程间同步等开销，渐渐成为高性能存储瓶颈。

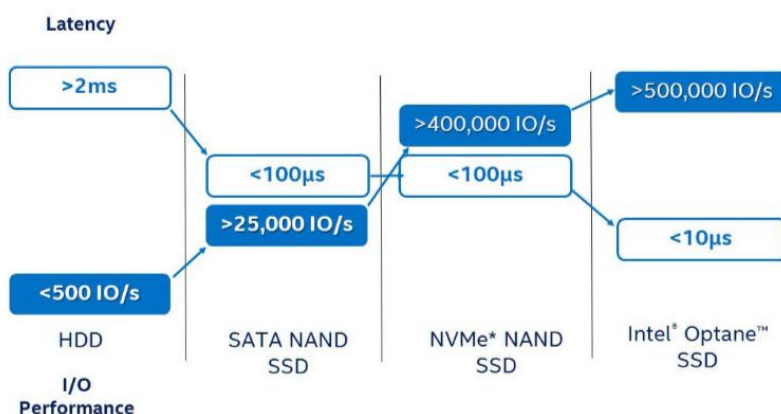
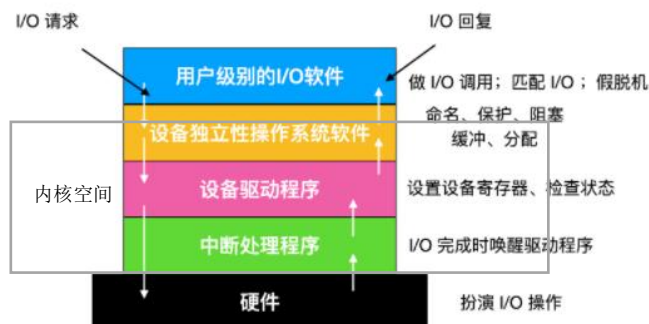


图 1-1 SSD 的发展

I/O 软件通常组织成四个层次，他们的大致结构如图 1-2 所示。



一次完整的 I/O 调用过程描述为：用户发起 I/O 请求，进行系统调用，然后由与设备无关的操作系统软件进行数据缓冲、块大小分配等操作，设备驱动程序接受读写请求后，对设备进行初始化，检查输入参数的有效性，并设置设备寄存器，最后进入中断处理程序，由硬件完成 I/O 读写，最后再通过中断唤醒驱动程

序并逐一向上返回，直至用户程序收到 I/O 恢复。这一系列操作中产生的开销包括数据拷贝、上下文切换以及多次中断，直接导致性能瓶颈。

其中，设备驱动程序主要对 I/O 设备进行控制，提供 I/O 设备到设备控制器的转换。为了能够访问设备的硬件，实际上也就意味着，设备驱动程序通常是操作系统内核的一部分（用户空间的设备驱动程序可能会干扰内核而造成崩溃）。应用程序和内核驱动模块的交互方式是，首先内核驱动模块需要在内核中，加载成功后，会被标识是块设备还是字符设备，同时定义相关的访问接口，包括管理接口、数据接口等。这些接口直接或间接和**文件系统子系统**结合，提供给用户态的程序，通过系统调用的方式发起控制和读/写操作。

1.2 用户态驱动

用户态驱动的出现可以减少上下文切换、系统调用等开销。在用户态，目前可以通过 UIO（Userspace I/O）或 VFIO（Virtual Function I/O）两种方式对硬件固态硬盘设备进行访问。

（1）UIO

UIO 框架最早于 Linux 2.6.32 版本引入，其提供了在用户态实现设备驱动的可能性。要在用户态实现设备驱动，主要需要解决以下两个问题。

➤ 如何访问设备的内存：Linux 通过映射物理设备的内存到用户态来提供访问，但是这种方法会引入安全性和可靠性的问题。UIO 通过限制不相关的物理设备的映射改善了这个问题。由此基于 UIO 开发的用户态驱动不需要关心与内存映射相关的安全性和可靠性的问题。

➤ 如何处理设备产生的中断：中断本身需要在内核处理，因此针对这个限制，还需要一个小的内核模块通过最基本的中断服务程序来处理。这个中断服务程序可以只是向操作系统确认中断，或者关闭中断等最基础的操作，剩下的具体操作可以在用户态处理。UIO 架构如图 1-3 所示，用户态驱动和 UIO 内核模块通过/dev/uioX 设备来实现基本交互，同时通过 sysfs 来得到相关的设备、内存映射、内核驱动等信息。

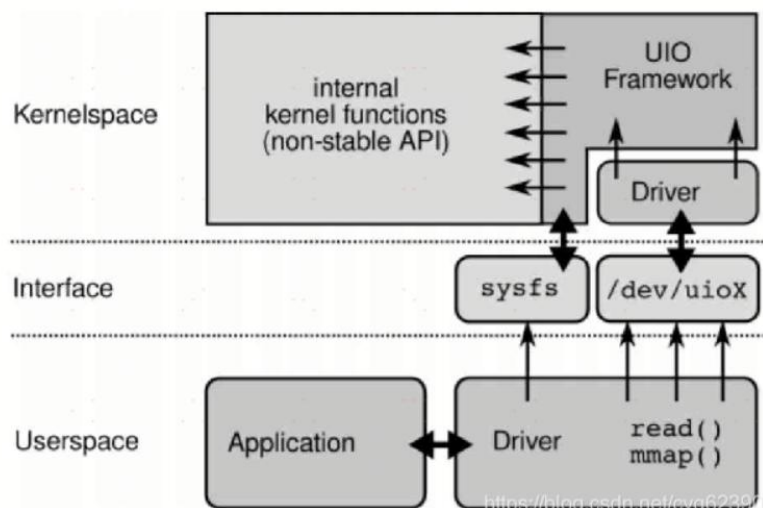
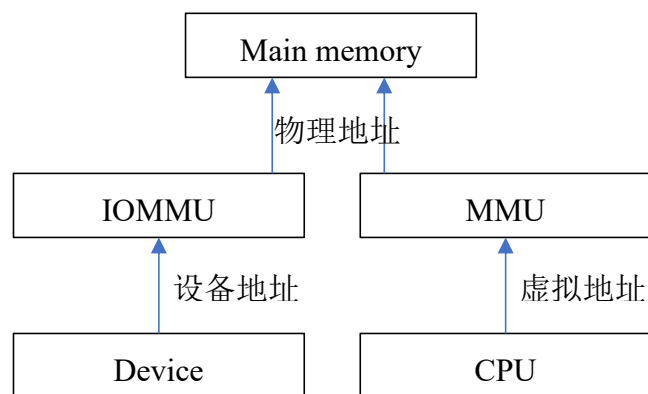


图 1-3 UIO 框架

(2) VFIO

相对于 UIO, VFIO 不仅提供了 UIO 所能提供的两个最基础的功能,更多的是从安全角度考虑,把设备 I/O、中断、DMA 暴露到用户空间,从而可以在用户空间完成设备驱动的框架。这里的一个难点是如何将 DMA 以安全可控的方式暴露到用户空间,防止设备通过写内存的任意页来发送 DMA 攻击。

IOMMU (I/O Memory Management Unit) 的引入对设备进行了限制,设备 I/O 地址需要经过 IOMMU 重映射为内存物理地址 (见下图)。那么恶意的或存在错误的设备就不能读/写没有被明确映射过的内存。操作系统以互斥的方式管理 MMU 和 IOMMU,这样物理设备将不能绕过或污染可配置的内存管理表项。



解决了通过安全可控的方式暴露 DMA 到用户空间这个问题后,用户基本上就可以实现使用用户态驱动操作硬件设备了。考虑到 NVMe SSD 是一个通用的 PCI 设备, VFIO 的 PCI 设备实现层 (vfiopci 模块) 提供了和普通设备驱动类似的作用,可高效地穿过内核若干抽象层,在 /dev/vfio 目录下为设备所在的 IOMMU group 生成相关文件,继而将设备暴露出来。

(SPDK 用户态驱动同时支持 UIO 和 VFIO 两种方式)

基于 UIO 或 VFIO，可以实现用户态的驱动，把一个硬件设备分配给一个进程，允许该进程来操作和读/写该设备。这在一定程度上提高了进程对设备的访问效率，不需要通过内核驱动来产生额外的内存复制，而是可以直接从用户态发起对设备的 DMA。

2、SPDK 介绍

SPDK (SPDK: A development kit to build high performance storage applications) 提供了一组工具和库，用于编写高性能存储应用程序或优化现有的存储系统。如图 2-1 所示，SPDK 主要以下四个主要组件：APP Scheduling、Drivers、Storage Services 和 Storage Protocols。APP Scheduling 提供了一个应用程序事件框架，用于利用 SPDK 的库来编写异步，轮询模式，无共享的服务器应用程序。Drivers 提供一个用户空间驱动程序，为程序提供了零拷贝，高度并行和直接访问 NVMe SSD 的功能。Storage Services 通过将由驱动程序导出的设备抽象化 (bdev)，并为存储应用程序提供用户空间块 I/O 接口。Storage Protocols 包含在 SPDK 框架上实现的加速应用程序，以支持各种不同的存储协议。

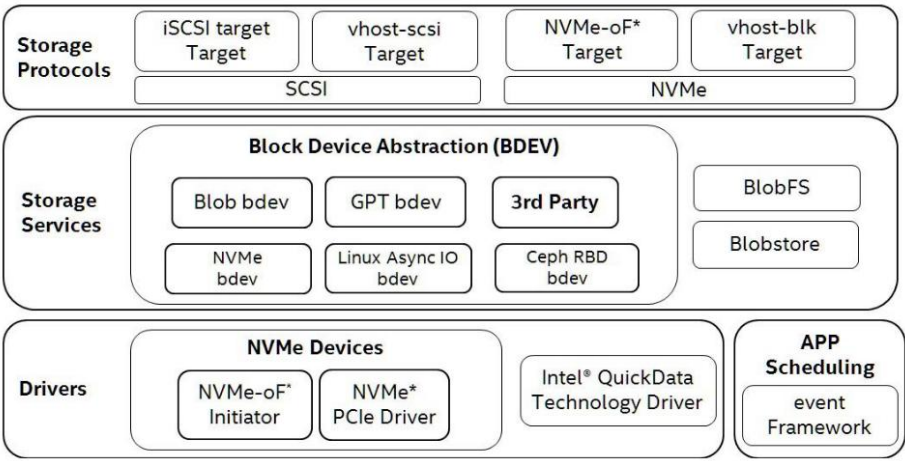


图 2-1 SPDK 框架

2.1 APP Scheduling

APP Scheduling 是 SPDK 提供的可选框架，用于编写异步、轮询模式、无共享的服务器应用程序。该框架主要有一下几个概念：reactors，event 和 poller。如图 2-2 所示，框架中每个 core 对应一个线程（即 reactor），并使用无锁队列连接线程，线程之间的以事件（events）的形式传递消息（在现代 CPU 架构中，消息传递比锁机制快很多）。

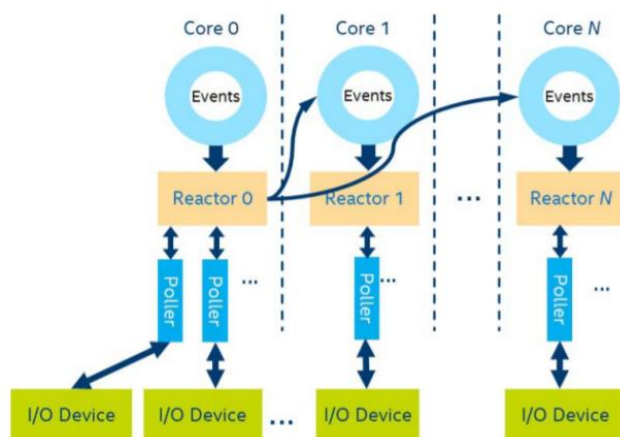


图 2-2 应用框架

① **events**: 为了实现线程之间的通信,并尽可能减少同步所带来的开销,此框架提供了以 **event** 进行的消息传递,每个 **event** 包含了一个绑定的函数指及其参数(异步执行,通过使用回调函数来发信号)。每个核心上运行着一个线程,这些线程被称为 **reactor**,它们的主要工作就是处理队列上到来的事件。

② **Reactor**: 本质上是一个线程以及一个无锁队列。来自任何内核的线程都可以将事件插入到任何其他内核的队列中。在每个内核上运行的 **reactor** 循环会检查传入的事件,并在收到事件时以先进先出的顺序执行它们。

③ **Pollers**: 与 **event** 一样,是带有参数的函数,但 **pollers** 会重复执行直到注销。**Reactor** 将 **pollers** 与其他的 **event** 交错执行,**pollers** 旨在轮询硬件以代替中断。

SPDK 的应用框架可以分为以下几个部分:(1)对 CPU core 和线程的管理;(2)线程间的高效通信;(3)I/O 处理模型以及数据路径(**data path**)的无锁化机制。

(1) 对 core 和线程的管理

SPDK 的一大宗旨是使用最少的 CPU 核和线程来完成最多的任务。为此,SPDK 在初始化程序时,通过调用 **spdk_app_start** 函数)来限定使用绑定 CPU 的哪些 cores,并且在每个核上运行一个 thread(即 **Reactor**)。**Reactor thread** 通过执行函数 **_spdk_reactor_run**(该函数主体包含一个 **while** 循环)不断轮询,直到 **Reactor** 的状态因 **spdk_app_stop** 函数的调用而改变。假设一个使用 SPDK 编程框架的应用运用了两个 CPU core,那么每个 core 上就会启动一个 **Reactor thread**。那么用户要如何执行自己的函数呢?

SPDK 提供了一个 **Pollers** 的概念,即用户定义函数的封装,在 **thread** 数据结构中由链表维护。在 **Reactor** 的 **while** 循环中,通过不停地检查 **Pollers** 的状态,进行相应的调用,用户定义的函数也因此可以进行相应的调用。由于单个 CPU 上只有一个 **Reactor thread**,所以同一个 **thread** 中不需要锁机制来保护资源。

而不同 CPU 的 core 上的 **Reactor thread** 是需要进行通信的,为了解决该问题,

SPDK 封装了线程间异步传递消息（Async Messaging Passing）的方式。

（2）线程间的高效通信

SPDK 放弃传统的加速方式来进行线程间的通信，因为这种方案比较低效。为了使同一个 thread 只处理自己所管理的资源，SPDK 提供了 Event（事件调用）机制。该机制的本质是每个 Reactor 对应的数据结构（struct spdk_reactor）维护了一个 Event 事件的 ring。这个环采用的是 MPSC 模型，任意线程可以将 Events 插入到 ring 中。（目前 SPDK 中的 Event ring 的缺省实现依赖于 DPDK 的机制，用线性锁的机制，相较于线程间采用锁的机制及逆行同步要高效的多）。

（3）I/O 处理模型以及数据路径的无锁化

SPDK 主要的 I/O 处理模型是 Run-to-completion，指运行直到全部完成。如若使用 SPDK 应用框架，天然符合该模型。但若不适用该框架，则需要编程者自己注意该事项。例如使用 SPDK 用户态 NVMe 驱动访问相应的 I/O QPair 进行读写操作（NVMe 的读写请求队列），SPDK 提供了异步读写的函数（spdk_nvme_ns_cmd_read），同时检查是否完成的函数（spdk_nvme_qpair_process_completions）。这些函数的调用应由一个线程完成，不应该跨线程处理。（SPDK 事件框架是对异步读写函数的封装）

SPDK 的 I/O 路径也采用无锁化机制。当多个 thread 操作同一 SPDK 用户态 block device (bdev) 时，SPDK 会提供一个 I/O channel 的概念（即 thread 和 device 的一个 mapping 关系）。不同的 thread 操作同一个 device 应该拥有不同的 I/O channel，每个 I/O channel 在 I/O 路径上使用自己独立的资源就可以避免资源竞争，从而去除锁的机制。（BDEV 中的概念，NVMe SSD 本身容量足够大，不同的应用程序可以共享该设备。还有一种用系统管理工具如 nvme-cli 进行管理多进程同时访问通过一 NVMe SSD）

总结一下，reactors 和 CPU core 以及 spdk thread 的关系如图 2-3 所示。

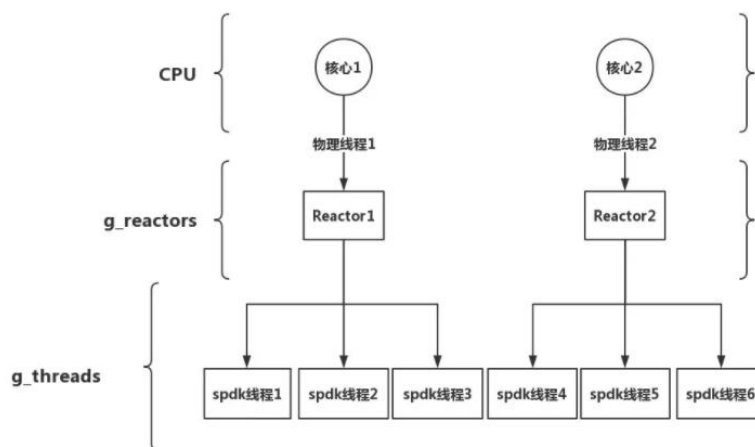


图 2-3 CPU cores、Reactors 和 thread 关系图

依靠 Events、Reactors 和 Pollers 以及上述机制，SPDK 应用框架通过轮询硬

件来代替中断，减少中断开销，同时用消息传递的方式代替锁机制，减少不同开销。

TODO：（io_device 和 io_channel 的解释 io_device 是设备的抽象，io_channel 是对该设备通道的抽象。一个线程可以创建多个 io_channel，io_channel 只能和一个 io_device 绑定，并且这个 io_channel 是别的线程使用不了的，也就是说 channel 和 thread 是意义对应的，所以无共享）

2.2 Drivers

NVMe 驱动程序是一个 C 库，可以直接链接到一个应用程序，该应用程序提供与 NVMe SSD 之间的直接零拷贝数据传输。它完全是被动的，这意味着它不会产生任何线程，只会响应来自应用程序本身的函数调用而执行操作。该库通过直接将 PCI BAR 映射到本地进程并执行 MMIO 来控制 NVMe 设备。

用户空间轮询模式 NVMe 驱动程序作为 SPDK 的核心部分，应用框架中的具体实现，消除了内核空间中的系统调用、数据拷贝的开销。同时，轮询可以避免软件中断，减少系统开销的同时可以让用户决定每个任务占用的 CPU 时间，而不是由内核调度决定，所以可以保证在有新的 IO 来临时，软件线程总是可用的。

SPDK 用户驱动除了基于背景中提到的 UIO 和 VFIO 的支持外，还通过分配 Hugepage（大页）减少缺页异常，从而减少缺页导致的开销（缺点：需要实先配置）。此外，还次啊用了以下优化方式来提供用户态驱动对设备的访问效率。

（1）异步轮询方式

前面提到的 UIO 和 VFIO 需要在内核中实现最基本的中断功能来响应设备的中断请求。而 SPDK 更进一步，直接通过异步轮询的方式来实现对设备完成状态的检测，进而避免了对中断的依赖。SPDK 用户态驱动的操作基本上都采用了异步轮询的方式，轮询到操作完成时会触发上层的回调函数，这使得应用程序无需等待读或写操作的完成。具体的对 NVMe SSD 设备的轮询可以参考 NVMe 规范。

（2）无锁化

内核态的驱动为了实现通用的块设备驱动，同时和内核其他模块深度集成，需要一些隔离的方法，比如信号量、锁、临界区等来保证操作的唯一性。SPDK 用户态驱动从性能优化的角度看，一个重要的优化点就是在数据通道上去掉对锁的依赖。为此 NVMe 的队列对不包含锁，给定的队列对一次只能由单个线程使用（扩展性好）。（NVMe 驱动程序不强制如此）。

需要主义的是 I/O 队列对占用的内存可以在主机内存中分配也可以放进控制器内存缓冲区，这取决于用户的使用方式和驱动。控制器

(3) NVMe 多进程

两种方式：

一、空间足够，提前划分。

二、空间不足，使用相关的管理工具，如 `nvme-cli` 工具，监控和配置管理 NVMe 设备。

2.3 Storage services

Storage services（存储服务）层包含 Blobfs/Blobstore 和用户空间块设备（BDEV）层。

其中 Blobstore 是位于 SPDK bdev 之上的 Blob 管理层，用于与用户态文件系统 BlobFS（Blobstore Filesystem）集成，从而代替传统的文件系统，支持更上层的服务，如数据库 MySQL、K-V 存储引擎 Rocksdb（**TODO：介绍**）等。

BlobFS 在管理文件时，主要依赖于 Blobstore 对 blob 的分配与管理。Blob 类似于文件的概念，而又不完全等同于文件，其并不支持所有文件的 POSIX 接口。BlobFS 与 Blobstore 的关系可以理解为 Blobstore 实现了对 Blob 的管理，包括 Blob 的分配、删除、读取、写入、元数据的管理等，而 BlobFS 是在 Blobstore 的基础上进行封装的一个轻量级文件系统，用于提供部分对于文件操作的接口，并将对文件的操作转换为对 Blob 的操作，BlobFS 中的文件与 Blobstore 中的 Blob 一一对应。在 Blobstore 下层，与 SPDK bdev 层对接。SPDK bdev 层类似于内核中的通用块设备层，是对底层不同类型设备的统一抽象管理，例如 NVMe bdev、Malloc bdev、AIO bdev 等。

接下来重点对 Blobstore（程序员编程）进行介绍。

(1) Blobstore 层次概念

Blobstore 层次结构如图 2-4 所示。

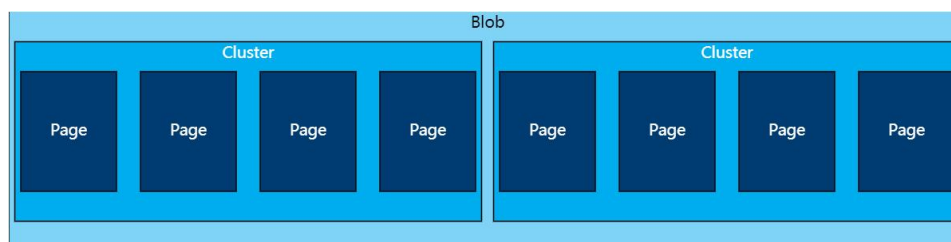


图 2-4 Blobstore 层次结构图

逻辑块：逻辑块由磁盘本身公开，磁盘编号从 0 到 N，其中 N 是磁盘中的块数。逻辑块通常是 512B 或 4KiB。

页面：页面定义为在 Blobstore 创建时定义的固定数量的逻辑块。组成页面的逻辑块始终是连续的。页面大小通常是 4Kib。

Cluster 群集：群集是在 Blobstore 创建时定义的固定页数。组成群集的页面

始终是连续的。群集也从磁盘的开头编号，其中群集 0 是第一组群集页面，群集 1 是第二组页面等。群集通常是 1MiB 大小，或 256 页。

Blob: blob 是一个有序的 Cluster 列表。应用程序主要操作对象，与 BlobFS 中的文件相对应。应用程序使用 Blobstore 提供的标识符来访问特定 blob。通过指定从 blob 开头的偏移量，以页为单位读取和写入 Blob。**BlobStore 中的 Cluster 不一定连续。**

Blobstore: 已由基于 Blobstore 的应用程序初始化的 SSD 称为“Blobstore”。Blobstore 拥有整个底层设备，该设备由私有 Blobstore 元数据区域和由应用程序管理的 blob 集合组成。

(2) Blobstore 的块管理

在 Blobstore 中 cluster0 是一个特殊的 cluster，用于存放 Blobstore 的所有信息和元数据，对每个 blob 数据块的查找，分配都是依赖于 cluster 0 所记录的元数据进行的。Cluster 0 的结构如下所示：

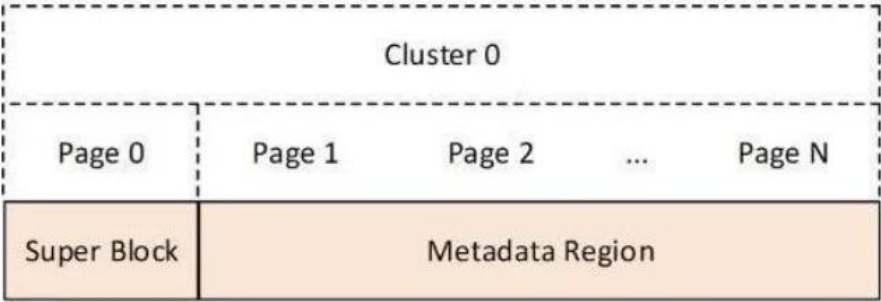


图 2-5 Cluster 0 结构

Cluster 0 中的第一个 page 作为 super block，Blobstore 初始化后的一些基本信息都存放在 super block 中，例如 cluster 的大小、已使用 page 的起始位置、已使用 page 的个数、已使用 cluster 的起始位置、已使用 cluster 的个数、Blobstore 的大小等信息。

Cluster 0 的其他 page 构成元数据域，元数据主要由一下几部分组成：

Metadata Page Allocation: 用于记录所有元数据页的分配情况。在分配或释放元数据页后，将会对 metadata page allocation 中的数据做相应的修改。

Cluster Allocation: 用于记录所有 cluster 的分配情况。在分配新的 cluster 或释放 cluster 后会对 cluster allocation 中的数据做相应的修改。

Blob Id Allocation: 用于记录 blob id 的分配情况。对于 blobstore 中的所有 blob，都是通过唯一的标识符 blob id 将其对应起来。在元数据域中，将会在 blob allocation 中记录所有的 blob id 分配情况。

Metadata Pages Region: 元数据页区域中存放着每个 blob 的元数据页。每个 blob 中所分配的 cluster 都会记录在该 blob 的元数据页中，在读写 blob 时，首先会通过 blob id 定位到该 blob 的元数据页，其次根据元数据页中所记录的信

息，检索到对应的 **cluster**。对于每个 **blob** 的元数据页，并不是连续的。

为了实现对磁盘空间的动态分配管理, Blobstore 中为每个 **blob** 分配的 **cluster** 并不是连续的。对于每个 **blob**，通过相应的结构维护当前使用的 **cluster** 以及 **metadata page** 的信息: **clusters** 与 **pages**。Cluster 中记录了当前该 **blob** 所有 **cluster** 的 **LBA** 起始地址, **pages** 中记录了当前该 **blob** 所有 **metadata page** 的 **LBA** 起始地址。最后，Blobstore 保证掉电不丢失数据（NVMe 这类 **bdev**）。

下面通过文件的读取来讲解 BlobFs 与 Blobstore 中的 I/O 流程。文件读取的流程图如图 2-6 所示。

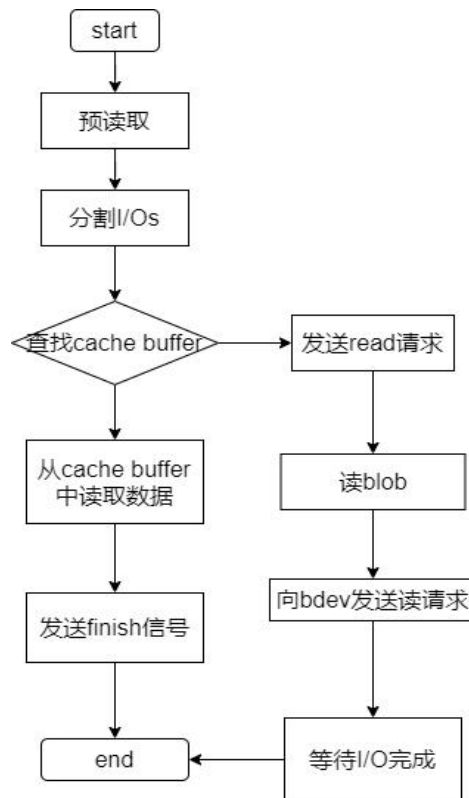


图 2-6 BlobFS 读流程

为了提高文件的读取效率，BlobFS 在内存中提供了 **cache buffer**。在文件读写时，首先会进行事先读取操作，将一部分数据从磁盘预先读取到内存的 **buffer** 中。其后，根据 **cache buffer** 的大小，对文件的 I/O 进行切分，使每个 I/O 的最大长度不超过一个 **cache buffer** 的大小。对于拆分后的文件 I/O，会根据其 **offset** 在 **cache buffer tree**（叶子节点为 **buffer node**）中查找相应的 **buffer**。若存在，则直接从 **cache buffer** 中读取数据，进行 **memcpy**。而对于没有缓存到 **cache buffer** 中的数据，将会对该文件的读取，转换到该文件对应的 **Blob** 进行读取。对 **Blob** 读取时候，根据已打开的 **blob** 结构中记录的信息，可以获取该 **blob** 所有 **cluster** 的 **LBA** 起始位置，并根据读取位置的 **offset** 信息，计算相应的 **LBA** 地址。最后向 **SPDK bdev** 层发送异步的读请求，并等待 I/O 完成。BlobFS 所提供的读操作

为同步读，I/O 完成后会在 `callback` 函数中，通过信号量通知 BlobFS 完成，至此文件读取结束。

实例：

TODO: (blobstore 代码解析 (直接对代码进行解释))

2.4 Storage Protocol

Storage Protocols 包含在 SPDK 框架上实现的加速应用程序，以支持各种不同的存储协议。例如，用于 iSCSI 服务加速的 iSCSI `taeget`，用于加速 VM 中的 `virtio-scsi/blk` 的 `vhost-scsi/blk target` 以及用于加速基于 NVMe over Fabric 协议服务的 NVMe-oF `target`。

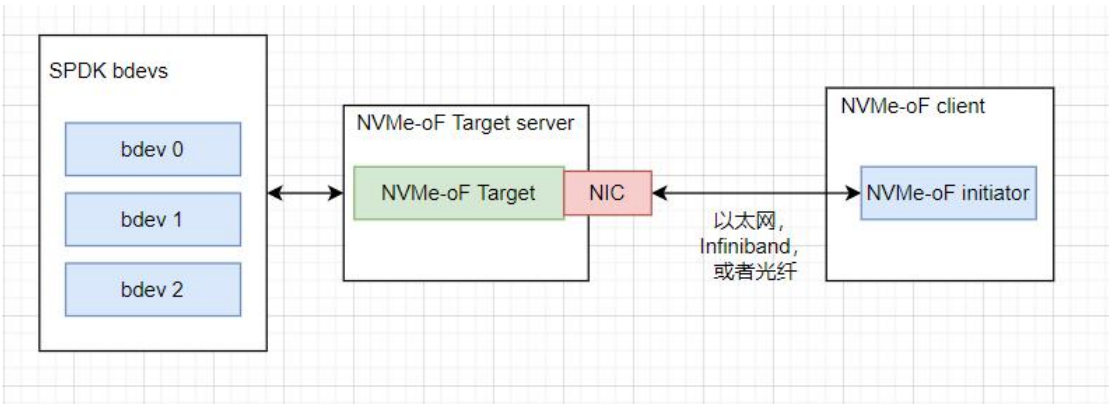


图 2-7 NVMe-of target/initiator

以 NVMe-of `target` 为例，如图 2-7 所示。NVMe-oF `target` 能基于 RDMA 通过网络呈现块设备。在用户空间 NVMe 驱动程序中，SPDK 提供了 NVMe-oF `initiator`，使得该驱动程序能够连接到远端 NVMe-oF `target`，并以与本地 NVMe SSD 相同的方式与它们进行交互。图 2-8 展示了 SPDK NVMe-oF `target` 的一般体系结构，NVMe-oF `target` 由 NVMe 库，NVMe `bdev` 以及 NVMe 用户空间驱动程序组成。其中，NVMe 库用于接收沿网络传输而来的请求，提取 NVMe 命令，并将其发送到 `bdev`，最后通过异步函数调用将结果返回给 NVMe-oF `initiator`。

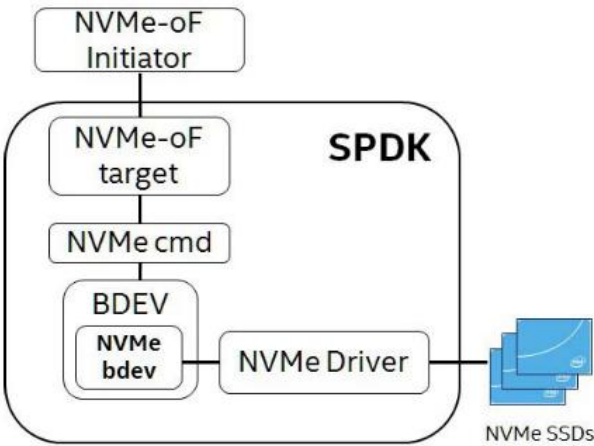


图 2-8 NVMe-oF target 结构

NVMe-oF target 作为 SPDK 的一大用处，在兼容性和通用性方面，一直在进步。

早期的 NVMe-oF 的规范中，只支持 RDMA 协议的网络传输，而在新的规范中，加入了基于 TCP/IP 协议的支持，所以在没有 RDMA 支持的网卡上也能运行，但性能不及 RDMA 网卡。而且如今的 NVMe-oF 协议能像 iSCSI target 一样，导出非 PCIe SSD，使得整个方案更加兼容。例如在 SPDK 中可以用 malloc 的 bdev 或者基于 libaio 的 bdev 来模拟出 NVMe 盘，将 NVMe 协议导入到 SPDK 通用 bdev 的语义，就能够给远端的用户呈现出 NVMe 盘。

SPDK 的 NVMe-of target 与 Linux Kernel NVMe-oF target 相比，具有以下优势：

- SPDK 的 NVMe-oF target 可以直接使用由 SPDK NVMe 用户态驱动封装的 bdev。相对于内核 NVMe 驱动而言，具有极大的性能优势。
- SPDK NVMe-oF target 完全采用 SPDK 所提供的编程框架，在所有 I/O 路径上都采用了无锁的机制，极大地提高了性能。
- SPDK 对于 RDMA 传输的实现融合了 SPDK 的编程框架，降低了在多并发连接情况下 I/O 处理的延迟。

3 SPDK 的配置和性能评估

3.1 SPDK 配置

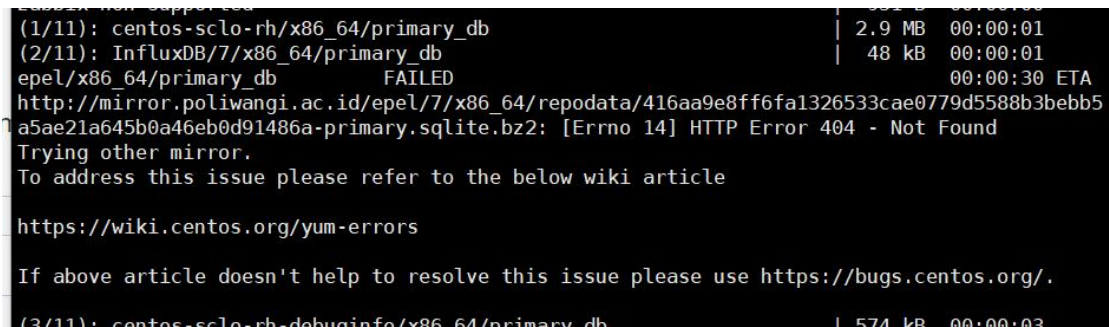
(1) 获取代码

```
git clone https://github.com/spdk/spdk
cd spdk
git submodule update --init
```

(2) 安装依赖

scripts/pkgdep.sh 脚本将自动安装构建 SPDK 所需的最低要求。

```
sudo scripts/pkgdep.sh
```



```
(1/11): centos-scllo-rh/x86_64/primary_db | 2.9 MB 00:00:01
(2/11): InfluxDB/7/x86_64/primary_db | 48 kB 00:00:01
epel/x86_64/primary_db FAILED 00:00:30 ETA
http://mirror.poliwangi.ac.id/epel/7/x86_64/repodata/416aa9e8ff6fa1326533cae0779d5588b3bebb5
a5ae21a645b0a46eb0d91486a-primary.sqlite.bz2: [Errno 14] HTTP Error 404 - Not Found
Trying other mirror.
To address this issue please refer to the below wiki article
https://wiki.centos.org/yum-errors
If above article doesn't help to resolve this issue please use https://bugs.centos.org/.
(3/11): centos-scllo-rh-debuginfo/x86_64/primary_db | 574 kB 00:00:03
```

Option -all 需的所有依赖项。

```
sudo scripts/pkgdep.sh -all
```

(3) building

Linux:

```
./configure
make
```

配置脚本有很多选项，可以通过运行一下命令来查看

```
./configure -help
```

请注意，并非默认情况下启用所有功能。例如，默认情况下不启用 RDMA 支持（因此不支持基于 Fabric 的 NVMe）。您可以通过执行以下操作启用它：

```
./configure --with-rdma
make
```

(4) Running the Unit Tests

脚本末尾的最后一条消息指示成功或失败。

```
./test/unit/unittest.sh
```

(5) 运行示例应用程序

在运行 SPDK 应用程序之前，必须分配一些大页面，并且所有 NVMe 和 IOAT 设备都必须与本机内核驱动程序解除绑定。SPDK 包含一个脚本，可以在 Linux

上自动执行此过程。该脚本应以 root 用户身份运行。它只需要在系统上运行一次

```
sudo scripts/setup.sh
```

要将设备重新绑定到内核，可以运行.

```
sudo scripts/setup.sh reset
```

默认情况下，脚本分配 2048MB 的大页面。要更改此数字，请按以下方式指定 HUGEMEM（以 MB 为单位）：

```
sudo HUGEMEM=4096 scripts/setup.sh
```

可以通过运行查看所有可用的参数

```
scripts/setup.sh help
```

示例代码位于 examples 目录中。这些示例将在构建过程中自动编译。只需调用任何不带参数的示例即可查看帮助输出。如果您的系统启用了 IOMMU，则可以以常规用户身份运行示例。如果不是，则需要以特权用户（root）身份运行。

在引用 spdk 在使用 spdk 之前，需要卸载内核中 NVMe 驱动程序并将指定的 NVMe 设备绑定到 Linux 内核中的 UIO 驱动程序上。在对使用到 SPDK 库的应用程序代码进行编译时，需要引用 SPDK 中 mk 文件夹下对应的链接库（主要是 spdk.common.mk 和 spdk.lib.mk），同时在运行时需要指定对应的配置文件，并配置文件中指定块设备名称等信息。

3.2 SPDK 的性能评估

关于 SPDK 的性能评估具体可参考论文，这里只简单的进行介绍。

在 NAND NVMe SSDs 上，利用 nvm perf 分别对内核 NVMe 驱动程序和用户空间 NVMe 驱动程序在使用单个 CPU core 和不同个数的 SSD 时，进行 4KB 的随机读取性能进行评估。测试结果如图 3-1。

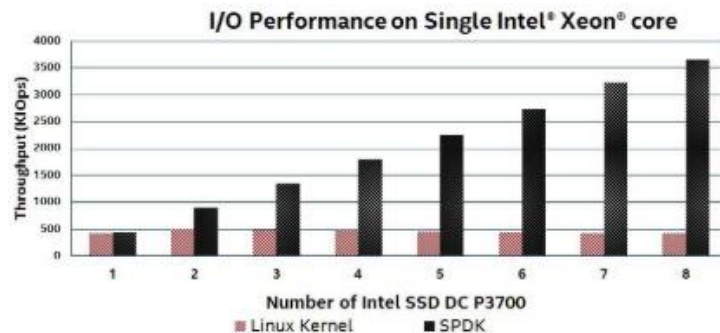


图 3-1 单个 core 的性能测试结果

可以发现 SPDK 用户空间 NVMe 驱动程序仅通过一个 CPU core 就能轻松释放 8 个 NVMe SSD 的性能。这得益于其无锁、无数据拷贝、无中断等机制导致

的极少的软件开销。

4 应用场景及其不足

从目前来讲，SPDK 并不是一个通用的适配解决方案。把内核驱动放到用户态，导致需要在用户态实施一套基于用户态软件驱动的完整 I/O 栈。文件系统毫无疑问是其中一个重要的话题，显而易见内核的文件系统，如 ext4、Btrfs 等都不能直接使用了。虽然目前 SPDK 提供了非常简单的文件系统 blobfs/blobstore，但是并不支持可移植操作系统接口（POSIX），为此使用文件系统的应用需要将其直接迁移到 SPDK 的用户态“文件系统”上，同时需要做一些代码移植的工作，如不使用可移植操作系统接口，而采用类似 AIO 的异步读/写方式。

目前 SPDK 使用比较好的场景有以下几种。

- 提供块设备接口的后端存储应用，如 iSCSI Target、NVMe-oF Target。
- 对虚拟机中 I/O 的加速。
- **SPDK 加速数据库存储引擎**，通过实现 RocksDB 中的抽象文件类，SPDK 的 blobfs/blobstore 目前可以和 RocksDB 集成，用于加速在 NVMe SSD 上使用 RocksDB 引擎，其实质是 bypass kernel 文件系统，完全使用基于 SPDK 的用户态 I/O 栈。

5 实例分析

以 `hello_blob.c` 为例，介绍一下 `blobstore` 的编程使用方式，重点在于理解异步逻辑和回调函数的设置使用。

`main()`:

```
int
main(int argc, char **argv)
{
    struct spdk_app_opts opts = {};
    int rc = 0;
    struct hello_context_t *hello_context = NULL;
    SPDK_NOTICELOG("entry\n");
    /* 应用编程框架中的初始化参数设置 */
    spdk_app_opts_init(&opts);
    opts.name = "hello_blob";
    opts.json_config_file = argv[1]; // 配置文件选择
    /*
     * 异步初始化 blobstore
     */
    hello_context = calloc(1, sizeof(struct hello_context_t));
    if (hello_context != NULL) {
        /*
         * spdk_app_start() will block running hello_start() until
         * spdk_app_stop() is called by someone (not simply when
         * hello_start() returns), or if an error occurs during
         * spdk_app_start() before hello_start() runs.
         */
        /* 创建并初始化 reactor，开始轮询，并向指定线程发送 hello_start 封装
        成的时间，直接返回*/
        rc = spdk_app_start(&opts, hello_start, hello_context);
        if (rc) {
            SPDK_NOTICELOG("ERROR!\n");
        } else {
            SPDK_NOTICELOG("SUCCESS!\n");
        }
        /* Free up memory that we allocated */
        hello_cleanup(hello_context);
    } else {
        SPDK_ERRLOG("Could not alloc hello_context struct!!\n");
        rc = -ENOMEM;
    }
    /* Gracefully close out all of the SPDK subsystems. */
    spdk_app_fini();
    return rc;
}
```

其中 `hello_start` 函数代码如下：

```

static void
hello_start(void *arg1)
{
    struct hello_context_t *hello_context = arg1;
    struct spdk_bdev *bdev = NULL;
    struct spdk_bs_dev *bs_dev = NULL;

    SPDK_NOTICELOG("entry\n");
    bdev = spdk_bdev_get_by_name("Malloc0");
    if (bdev == NULL) {
        SPDK_ERRLOG("Could not find a bdev\n");
        spdk_app_stop(-1);
        return;
    }
    bs_dev = spdk_bdev_create_bs_dev(bdev, NULL, NULL); // 从抽象出的 bdev
    创建 blobstore
    if (bs_dev == NULL) {
        SPDK_ERRLOG("Could not create blob bdev!!\n");
        spdk_app_stop(-1);
        return;
    }
    spdk_bs_init(bs_dev, NULL, bs_init_complete, hello_context); // 初始
    化 bs, 返回前执行回调函数 bs_init_complete
}

```

bs_init_complete 函数代码如下所示:

```

static void
bs_init_complete(void *cb_arg, struct spdk_blob_store *bs,
                 int bserrno)
{
    struct hello_context_t *hello_context = cb_arg;
    SPDK_NOTICELOG("entry\n");
    if (bserrno) {
        unload_bs(hello_context, "Error init'ing the blobstore",
                  bserrno);
        return;
    }
    hello_context->bs = bs;
    SPDK_NOTICELOG("blobstore: %p\n", hello_context->bs);
    /*
     * 需要申请的 buffer 空间大小, 保存下来
     */
    hello_context->io_unit_size = spdk_bs_get_io_unit_size(hello_context->bs);
    /*
     * 创建一个 blob, 可以通过使用 spdk_thread_send_msg() 函数来控制处理时间
     */
    create_blob(hello_context);
}

```

接下来同样通过设置回调函数, 如图 5-1 绘制的流程图, 依次执行 blob 的创建、写入、读取和卸载。

其中, 带有 spdk 前缀的 spdk_xxx 的函数在调用是都需要传入响应的 callback 函数。

spdk_bs_create_blob: 通过调用_spdk_bs_create_blob 函数来获取 blobid, 设

置元数据，创建 blob 结构并设置 blob 状态。

spdk_bs_open_blob: 通过调用 `_spdk_bs_open_blob`，根据 blobid 索引到对印的元数据页从而定位到其位置，再调用 `_spdk_blob_load` 从硬盘中加载该 blob。

spdk_blob_resize: 用于 `resize` 的大小，最小单元为 `cluster`。

spdk_blob_sync_md: 为了加速性能，spdk 再 DRAM 中缓存了 blobstore 的元数据，所以再每次修改元数据的时候最好都能够主动 `sync` 一下。

blob_write: 申请 `write_buffer` 以及专属的 `io_channel`。

spdk_blob_io_write: 调用 `blob_request_submit_op` 函数，根据 `op` 类型配置写入参数，最终通过 `channel->dev->write` 函数经由绑定的 `channel` 写入到抽象块设备 `dev` 中。

spdk_blob_io_read: 与 `spdk_blob_io_write` 类似。

spdk_blob_close: 关闭该 blob，引用个数减一。

spdk_bs_delete_blob: 从 blobstore 链表中删除 blob。

