Extending a File System to Support NVMe Devices with SPDK

Ao Li
University of Toronto

Geoffrey Yu
University of Toronto

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse egestas arcu sit amet libero imperdiet fringilla. Nunc tempus libero vitae sapien pretium, id gravida tortor vestibulum. Suspendisse ut velit sed mi vehicula sollicitudin eu quis urna. Nulla sed libero eu enim fringilla eleifend ac at augue. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et dapibus turpis. Vestibulum fringilla nibh a enim tempus lobortis. Ut posuere risus eu mi consequat, eget pretium nibh varius.

1 Introduction

Lorem ipsum [1] dolor sit amet, consectetur adipiscing elit. Suspendisse egestas arcu sit amet libero imperdiet fringilla. Nunc tempus libero vitae sapien pretium, id gravida tortor vestibulum. Suspendisse ut velit sed mi vehicula sollicitudin eu quis urna. Nulla sed libero eu enim fringilla eleifend ac at augue. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et dapibus turpis. Vestibulum fringilla nibh a enim tempus lobortis. Ut posuere risus eu mi consequat, eget pretium nibh varius.

Donec maximus purus nec elit dapibus ultricies nec vitae mi. Pellentesque vel turpis vitae mi sodales volutpat. Nullam vehicula neque eu ipsum pretium, eget aliquet nibh convallis. Proin iaculis, tortor pulvinar ultrices dictum, magna massa imperdiet arcu, ac bibendum velit arcu eu odio. Vivamus vitae lectus porta, tristique diam eu, vestibulum dolor. Maecenas dapibus porttitor cursus. Nunc vel quam eu mi fringilla congue. Vestibulum fermentum sagittis arcu, ut pharetra eros eleifend nec. Nam nisl augue, efficitur in porta eget, dapibus eget mauris. Vivamus sit amet sapien faucibus, elementum odio et, pulvinar purus.

2 Background

In this section we present the background of NVMe interface and SPDK framework.

2.1 NVMe and I/O Queue

NVMe [2] is an open interface specification designed to allow host software to communicate with a non-volatile memory subsystem (NVM) via a peripheral component interconnect express (PCIe) bus. Previous standards such as serial-attacked SCSI and serial advanced technology attachment can handle queue depths of 254 and 32 respectively. NVMe is able to handle queue depths of up to 65535 I/O queues with up to 64 Ki outstanding commands per I/O queue, which allows an NVMe device to support parallel operations. An I/O queue is composed of a submission queue and a completion queue. Host software issues I/O commands to a submission queue and completions are placed into the associated completion queue by the controller. Note that the order of completions is not determined by the submission oder of the commands.

2.2 SPDK and Block Devices

SPDK [1] is an open source library that allows developer to implement high performance, scalable, user-mode storage applications. A block device in SPDK is an abstraction of all block devices, where I/O commands are processed and issued to corresponding physical block devices such as NVMe devices and Malloc devices. SPDK provides a event framework, where different threads to exchange data through passing messages to one another. It allows a user to build asynchronous, lockless, and high performance applications.

For each thread, SPDK uses an io channel to represent the channel for accessing an I/O device. I/O requests issued to the block device will be forwarded to the underlying physical device. In our implementation, io channel corresponds to I/O queue of the underlying NVMe device, the framework builds I/O commands based on I/O requests and submits them to submission queue. It then polls for I/O completion on each queue pair with outstanding I/O to receive completion callbacks. Figure 1 provides a graphical representation of a host application using SPDK block devices to interact with an NVMe device. In the host application, each thread submit

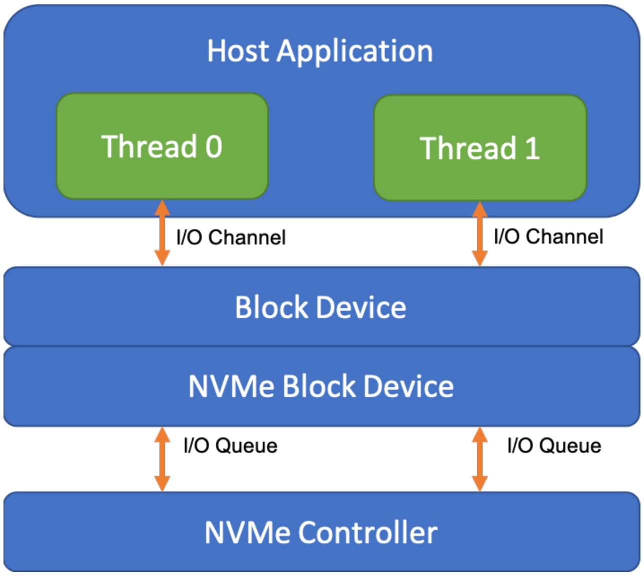

Figure 1: Example of an SPDK application.

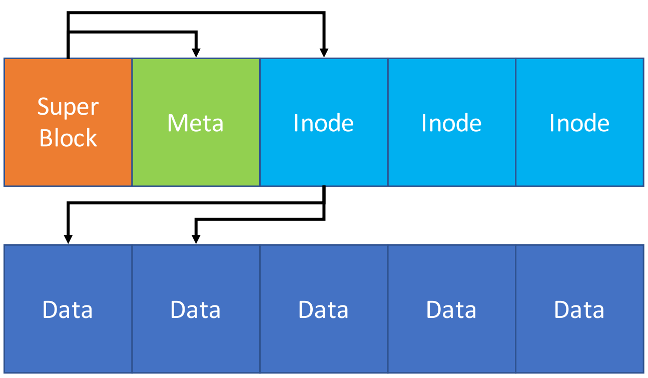

Figure 2: The layout of blocks with different types in testFS.

I/O requests to its corresponding I/O channel and the I/O channel forward the I/O requests to the actual physical device based on the implementation of the physical block device. The framework invokes the callback function when the I/O request is finished.

2.3 TestFS

TestFS [3] is a user space file system which is similar to EXT3, a journaled file system that is commonly used by the Linux kernel. TestFS has three levels of indirections, which is illustrated in Figure 2. A super block points to meta data and inode block of the root directory. The meta data store the freemap of blocks as well as the checksum of the data blocks. Inode blocks point to data blocks where the actual data are stored. In testFS, both directory data and indirect inode data are stored in data blocks. The space for inode blocks and data blocks are pre-allocated and fixed during the life cycle of testFS.

Note that our file system is based on testFS because testFS is well maintained, user level, and well documented file system, that allows us to build a proof of concept quickly. We believe our modifications to testFS can be applied to most of file systems.

3 Project Overview

In this section we provide an overview of our project by enumerating our goals, describing the primary challenges we faced, and highlighting the key ideas behind our approach to addressing these challenges to meet our goals.

3.1 Project Goals

The high level goal of our project is to explore the feasibility and performance implications of enabling parallel access to NVMe storage devices in a user space file system. We work toward this goal by integrating SPDK with testFS. More concretely, in our project we aim to:

1. Demonstrate the feasibility of using SPDK to interact with NVMe devices in testFS.

2. Modify the testFS write path to leverage the ability to make asynchronous block writes with multiple threads.

3. Quantify the performance improvements of an asynchronous write path by benchmarking our implementation.

3.2 Key Challenges

Challenge 1: Callbacks and Synchronous Code. Reads and writes to block devices with SPDK are asynchronous operations. SPDK uses callback functions as the mechanism for scheduling code that should run after an asynchronous operation has completed. However testFS is designed with synchronous operations in mind. The file system code is written with the assumption that, after a thread makes an I/O request, the thread will wait until the request completes before executing any additional code. Ultimately, this combination of synchronous code and a library that uses asynchronous callbacks makes it difficult to integrate testFS with SPDK without making invasive changes throughout the testFS codebase.

Challenge 2: Lock Contention. Our project is a proof of concept for next generation file systems that may use hundreds of threads. To ensure the scalability of our ideas, we need to avoid designs that have multiple threads acquiring the same locks to prevent contention among the threads.

3.3 Our Approach: Key Ideas

The key idea behind our approach is to leverage a multi-threaded architecture where each thread has a specific purpose. Specifically, we use three threads: (i) a control thread responsible for executing the file system logic and for interacting with the user through the testFS REPL, (ii) a metadata thread responsible for submitting I/O requests for blocks that are used for file system metadata, and (iii) a data thread responsible for submitting I/O requests for data blocks. This architecture allows us to achieve asynchronous writes with multiple threads without incurring a significant engineering overhead.

Reconciling Callbacks and Synchronous Code. A multi-threaded architecture allows us to use existing thread synchronization techniques as a way to be able to wait for asynchronous requests to complete. The threads responsible for issuing I/O requests are distinct from the control thread, which means callbacks only need to be registered on the lowest level code responsible for issuing read/write requests with SPDK. When asynchronous requests complete, the callback will execute on the I/O request thread and can then "signal" the control thread as needed. We discuss two techniques for thread synchronization in Section 4.1. Ultimately this approach allows us to avoid adding callback functions throughout the entire testFS codebase and therefore allows us to avoid making invasive changes to the file system.

Avoiding Lock Contention. To avoid lock contention, we use thread synchronization techniques that either (i) do not require locks, or (ii) have a lock per thread, to prevent contention among I/O threads. We discuss our proposed thread synchronization techniques in greater detail in Section 4.1.

4 Implementation Details

To implement asynchronous writes with multiple threads in testFS, we added utilities to facilitate thread synchronization and we modified the file system's write path to make it more amenable to asynchronous writes. For a mapping of the key changes to the relevant source code files, please see Appendix B.

4.1 Thread Synchronization

In our multi-threaded architecture, the control thread needs to be able to wait for asynchronous I/O requests to complete. We accomplish this through thread synchronization using shared objects: a primitive future or several semaphores. We describe both approaches, however for the remainder of the report we only reference the primitive future technique as both synchronization techniques accomplish the same thing from a concurrency control standpoint.

4.1.1 Primitive Futures

TODO

4.1.2 Semaphores

TODO

4.2 Asynchronous I/O

To implement asynchronous I/O, we added asynchronous analogues of the `read_blocks()` and `write_blocks()` functions called `read_blocks_async()` and `write_blocks_async()`. These `async` functions accept the same arguments as their synchronous counterparts. However callers of these `async` functions also need to pass in a pointer to a future and the thread ID on which the I/O request should be handled. This allows the control thread to direct metadata block requests and data block requests to distinct threads.

When these asynchronous read/write functions are called, the future's `expected_count` for the handling thread is first incremented and then the request is sent to the handling thread using SPDK's message passing library. Upon receiving this message, the handling thread then submits the I/O request to the underlying NVMe device using its dedicated I/O channel. When the request completes, a callback is executed on the handling thread. The callback function increments the future's `count` for the handling thread to "notify" the calling thread that the request has completed. Since the I/O request is asynchronous, the calling thread is free to do other work while the I/O request is being handled. When the calling thread needs to wait for the I/O request to be completed it can call `spin_wait()` on the future.

To be able to compare asynchronous I/O requests and synchronous I/O requests, we modified `read_blocks()` and `write_blocks()` so that they can work with SPDK as well. These functions were implemented by calling their asynchronous counterparts and then immediately waiting on the future. This ensures that the I/O request completes before the function returns.

4.3 Write Path Modifications

TODO

5 Evaluation

6 Conclusion

Acknowledgments

We would like to thank Shehbaz Jaffer for his guidance and feedback on our project. We also thank the Systems and Networks Lab at the University of Toronto for providing access to hardware for our experiments.

References

[1] SPDK. https://spdk.io.

[2] Specifications NVMe. https://nvmexpress.org/resources/specifications/.

[3] TestFS filesystem. https://github.com/shehbazj/testfs.

A Reproducing our Results

B Code Walkthrough