SANBlaZe

White Paper

Verification of ZNS

Zoned Namespaces

Verification for SSD Drives

Authors

Vince Asbridge, Founder & President, SANBlaze

Haiyan Lin, Sr. Software Engineer, SANBlaze

Table of Contents

List of Figures

1. Introduction

SANBlaze has announced the availability of ZNS (Zoned Namespace) verification that allows you to quickly and effectively test and validate the ZNS implementation of your solid state drives (SSDs). This white paper introduces ZNS and describes how to verify that your SSDs have implemented all ZNS features correctly using the SANBlaze SBExpress test and validation system.

2. Understanding ZNS

NVMe™ Zoned Namespace (ZNS) is a technical proposal under standardization by the NVM Express™ organization. It divides the logical address space of a namespace into zones. Each zone provides a Logical Block Address (LBA) range that must be written sequentially and if written again must be explicitly reset. This operation principle allows created namespaces that expose the natural boundaries of the device and provides offload management of internal mapping tables to the host.

2.1 Why ZNS for SSDs?

SSDs are intrinsically zoned devices due to flash characteristics. A page is the smallest area of the NAND flash memory that supports a write operation and consists of all the memory cells on the same WordLine. An erase block is the smallest area of the flash memory that can be erased in a single operation. Page and block sizes differ per manufacturer and flash generation. For example, 19nm 64Gb MLC flash contains 16KB page size and 4MB block size. 16KB page size corresponds to 16,384 bytes that are dedicated for data and 1,280 bytes that are available for control and Error Correction Code (ECC) information.

NAND flash technology has evolved from SLC (Single-Level Cell, one bit per cell) to MLC (Multi-Level Cell, 2 bits per cell), then to TLC (3 bits per cell) and the current QLC (4 bits per cell). SLC NAND provides faster write speed and longer write endurance (around 30,000 – 50,000 Program/Erase Cycles) but is more expensive. MLC NAND offers a larger capacity, twice the density of SLC but with less endurance (around 3,000 Program/ Erase Cycles). TLC and QLC increase capacity significantly but at the cost of much less endurance (maybe around 300 Program/Erase Cycles), lower performance, and the need for more DRAM to map the higher capacity. DRAM is the highest cost after NAND in a typical SSD.

ZNS introduces a new type of NVMe drive that provides several benefits over traditional SSDs. It divides one namespace into multiple zones and only allows sequential write in each zone.

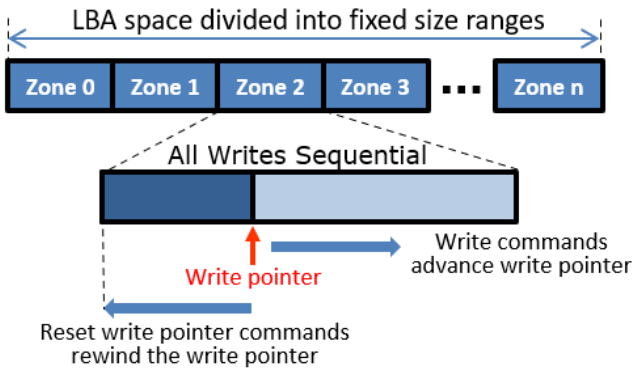

Figure 1: Zoned Namespace

SSDs cooperate using distributed FTL for the sequential access and eliminate multiple layers of indirection. No complex topology provisioning is needed because zones are logical. ZNS reduces write amplification, improves internal data movement, improves wear reduction, improves latency outliers and throughput, reduces DRAM in SSD (smaller L2P) and reduces the need for media over-provisioning. With the zones aligned to the internal physical properties of the NAND flash, several inefficiencies in the placement of data can be eliminated. In particular, the problem commonly known as the log-on-log challenge is naturally solved.

2.2 ZNS Model and State Machine

The ZNS model is similar to ZBC (Zoned Block Commands) and ZAC (Zoned ATA Commands) for SMR HDDs, but the interface is optimized for SSDs to align with media characteristics (i.e., aligned fixed zone size to NAND block sizes, and aligned variable zone capacity to physical media sizes). There are 7 states defined for ZNS as well: Empty, Full, Implicit Open, Explicit Open, Closed, Read Only and Offline. Valid transitions between each state can be changed by the NVMe Write, Zone Management Command (Open, Close, Finish, Reset) and Device Resets as shown in the zone state machine below.

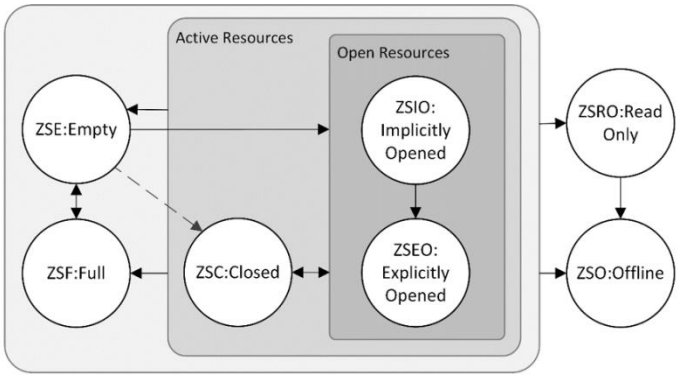

Figure 2: Zone State Machine

2.3 ZNS Commands

ZNS commands include Zoned Admin Command Sets and Zoned I/O Commands.

2.3.1 Zoned Admin Command Sets

The *NVMe – TP 4053 Zoned Namespaces 2020.03.19 – Final* specification provides specific additions to the ZNS Admin Command Set as follows:

- Identify Namespace Data Structure (TBD – specification not complete)
- Identify Controller Data Structure (TBD – specification not complete)
- Asynchronous Events Information
- Log page 0xBF
- Set Feature (Asynchronous Event Configuration)
- Sanitize
- Controller Architecture (Administrative Controller)

2.3.2 Zoned I/O Commands

The *NVMe – TP 4053 Zoned Namespaces 2020.03.19 – Final* specification provides specific commands for the Zoned Namespaces Command Set as follows:

- Flush
- Write
- Read
- Write Uncorrectable
- Compare
- Write Zeroes
- Dataset Management
- Verify
- Reservation Register
- Reservation Report
- Reservation Acquire
- Reservation Release
- Copy
- Zone Management Send
- Zone Management Receive
- Zone Append

Most commands are defined in the *NVMe specification v1.4* except the "Zone Management Send," "Zone Management Receive" and "Zone Append" which are new.

Each zone is allowed to sequentially write only. If a sequential write in one zone in an SSD has a Queue Depth > 1 then it means multiple writes per zone, and it will involve significant lock contention and affect write performance. The Benchmark below shows multiple writes to a zone has low scalability, and one write per zone generates good performance. But write performance is improved by writing to multiple zones. Using the "Zone Append" command that appends data to a zone with an implicit write pointer (without defining the offset) improves performance significantly. The SSD returns an LBA where data was written in the zone and it will allow a higher Queue Depth (no host serialization).

Figure 3: Scalability with Multiple Writers

3. ZNS Verification by SANBlaze

The SANBlaze engineering team has incorporated ZNS testing into its SBExpress platform, and we are proud to be the industry's first to provide ZNS testing and validation to our customers. SANBlaze Application Support for ZNS includes Certified by SANBlaze pre-developed test cases that allow users to start validating ZNS support and capability right out of the box. Test cases support the following functionality:

- Support all Zoned Admin Command Sets and Zoned I/O Command Sets defined in the *NVMe – TP 4053 Zoned Namespaces 2020.03.19 – Final* specification in our SBExpress GUI, command line interface, XML API interface, and Python wrapped API interface for test automation.
- Customized Linux driver to handle ZNS state machine transition and sequential write requirement in each zone.
- Support multiple threads I/O running in the zones of ZNS in parallel with high throughputs. Each zone can be tested using write, read, compare, and append as needed. Each zone will be reset at the start, and then later when finished at the end.
- Namespace management for ZNS.
- Negative testing through scripts to test all ZNS features.

3.1 Zone Management/Append Examples with the SANBlaze Platform

3.1.1 Zone Management Receive

```
c:\LIN\Sanblaze\souce\vlun1_tip\factory_default_virtualun\apis>py -3
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from sanblaze_apis import XML_API
>>> t107=XML_API(tester_IP="192.168.100.111", port=0, target=107, lun=1)
>>> t108=XML_API(tester_IP="192.168.100.111", port=0, target=108, lun=1)
>>> help(t108.zone_management_receive)
Help on method zone_management_receive in module sanblaze_apis:

zone_management_receive(start_LBA=0, return_bytes=4096, receive_action=0) method of sanblaze_apis.XML_API instance
    The zone_management_receive() function returns a data buffer that describes information about zones.
    start_LBA = the location of a data buffer where data is transferred from (Recommend to put hex number like 0x00)
    return_bytes = number of bytes return (Recommend to put hex number like 0x1000 which is 4096 bytes)
    receive_action = zone receive action specific features (Recommend to put hex number like 0x100 to list all ZSE zones)
        Bits    Description
        31:17       Reserved
        16      Zone Receive Action Specific Features:
            1h  Return the Number of Zones field in the Report Zones data structure and in the Extended Report Zone data structure
                indicates the number of fully transferred zone descriptors in the data buffer.
            0h  Return the Number of Zones field in the Report Zones data structure and Extended Report Zone data structure indicates
                the number of zone descriptors that match the criteria in the Zone Receive Action Specific field.
        15:08       Zone Receive Action Specific Field:
            0h      List all zones.
            1h      List the zones in the ZSE:Empty state.
            2h      List the zones in the ZSIO:Implicitly Opened state.
            3h      List the zones in the ZSEO:Explicitly Opened state.
            4h      List the zones in the ZSC:Closed state.
            5h      List the zones in the ZSF:Full state.
            6h      List the zones in the ZSRO:Read Only state.
            7h      List the zones in the ZSO:Offline state.
            8h-FFh  Reserved
        07:00       Zone Receive Action (ZRA):
            00h     Report Zones: Reports zone descriptor entries through the Report Zones data structure.
            01h     Extended Report Zones: Reports zone descriptor entries through the Extended Report Zones data structure. This value
                is supported if the namespace is formatted with a non-zero Zone Description Extension Size. Otherwise, the command shall
                be aborted with a status code of Invalid Field in Command.
            02h-FFh Reserved
```

Figure 4: Help Info for Zone Management Receive

```
>>> t108.zone_management_receive()    # List all zones with default input arguments
Command ZoneManagementReceive passed on port 0 target 108 in tester 192.168.100.111. Output is
decoded as follows:
                                    Num_Zones = 0x0000000000003B98
                Zone_Descriptor_0:
                                    Zone_Type = 0x02
```

```
                                    Zone_State = 0x10
                                Zone_Attributes = 0x00
                                  Zone_Capacity = 0x0000000000003000
                                 Zone_Start_LBA = 0x0000000000000000
                                  Write_Pointer = 0x0000000000000000
                Zone_Descriptor_1:
                                    Zone_Type = 0x02
                                    Zone_State = 0x10
                                Zone_Attributes = 0x00
                                  Zone_Capacity = 0x0000000000003000
                                 Zone_Start_LBA = 0x0000000000004000
                                  Write_Pointer = 0x0000000000004000
. . .

>>> t108.zone_management_receive(receive_action=0x10000) # Report zone structure in data buffer
Command ZoneManagementReceive passed on port 0 target 108 in tester 192.168.100.111. Output is
decoded as follows:
                                    Num_Zones = 0x000000000000003F
                Zone_Descriptor_0:
                                    Zone_Type = 0x02
                                    Zone_State = 0x10
                                Zone_Attributes = 0x00
                                  Zone_Capacity = 0x0000000000003000
                                 Zone_Start_LBA = 0x0000000000000000
                                  Write_Pointer = 0x0000000000000000
. . .
```

3.1.2 Zone Management Send

```
>>> help(t108.zone_management_send)
Help on method zone_management_send in module sanblaze_apis:

zone_management_send(start_LBA=0, send_action=3) method of sanblaze_apis.XML_API instance
    The zone_management_send() function performs an action on one or more zones.
    start_LBA = the lowest LBA of the zone the command operates on (Recommend to put hex number like 0x00)
    send_action = zone send action (Recommend to put hex number like 0x104 to reset all zones)
        Bits    Description
        08      Select All: If the bit is cleared to '0', then the start_LBA field specifies the lowest logical block of the zone.
                If the bit is set to '1', then the SLBA field shall be ignored.
        07:00
            00h     Reserved
            01h     Close Zone: Close one or more zones.
            02h     Finish Zone: Finish one or more zones.
            03h     Open Zone: Open one or more zones.
            04h     Reset Zone: Reset one or more zones.
            05h     Offline Zone: Offline one or more zones.
            06h-0Fh Reserved
            10h     Set Zone Descriptor Extension: Attach Zone Descriptor Extension data to a zone in the ZSE:Empty state and transition
                the zone to the ZSC:Closed state.
            11h to FFh      Reserved
```

Figure 5: Help Info for Zone Management Send

```
>>> t108.zone_management_send()       # open zone 0 with default input arguments
Command ZoneManagementSend passed on port 0 target 108 in tester 192.168.100.111. Output is
decoded as follows:

Command Completion Queue Status is decoded as follows:
                                CommandSpecific = 0x00000000
                                      Reserved0 = 0x00000000
                                SQ_Head_Pointer = 0x0004
                                  SQ_Identifier = 0x0001
                              Command_Identifier = 0x07CB
                        Status_Field:
                                     PhaseBit = 0x01
                                    StatusCode = 0x0000
                                 StatusCodeType = 0x00
                                       Reserved = 0x00
                                MoreInformation = 0x00
                                     DoNotRetry = 0x00

>>> t108.zone_management_receive()  # List all zones with default input arguments
Command ZoneManagementReceive passed on port 0 target 108 in tester 192.168.100.111. Output is
decoded as follows:
                                    Num_Zones = 0x0000000000003B98
```

```
                Zone_Descriptor_0:
                                      Zone_Type = 0x02
                                     Zone_State = 0x30
                                 Zone_Attributes = 0x00
                                   Zone_Capacity = 0x0000000000003000
                                  Zone_Start_LBA = 0x0000000000000000
                                   Write_Pointer = 0x0000000000000000
                Zone_Descriptor_1:
                                      Zone_Type = 0x02
                                     Zone_State = 0x10
                                 Zone_Attributes = 0x00
                                   Zone_Capacity = 0x0000000000003000
                                  Zone_Start_LBA = 0x0000000000004000
                                   Write_Pointer = 0x0000000000004000
. . .
```

3.1.3 Zone Append

```
>>> help(t108.zone_append)
Help on method zone_append in module sanblaze_apis:

zone_append(start_LBA=0, num_LBAs=1, data_pattern=165, pattern_from_fileName=None, block_size=None, limited_retry=0, force_unit_access=0, protection_info=None,
 protection_info_remap=0, ref_tag=None, app_tag=None, app_tag_mask=None) method of sanblaze_apis.XML_API instance
    The zone_append() function is used to write data and metadata. Data placement inside the zone is done by the controller so that the
        data is contiguously placed from the zone start LBA of the zone to the end of the writable capacity of the zone.
    start_LBA = the first LBA for zone append (Recommend to put hex number like 0x00)
    num_LBAs = the number of LBAs to zone append (Recommend to put hex number like 0x01)
    data_pattern = data pattern ([numbytes] byte of data) used for zone append
    pattern_from_fileName: Data pattern file name to read from the tester in directory /virtualun/lundata/initiator/ and it is used for
        zone append, e.g. pattern_from_fileName="temp.bin". It has higher priority than input argument data_pattern.
    block_size: In general user does not have to specify and it will use the namespace formated block size by default, e.g.
        block_size=None. If need to specify please enter hex number like 0x1000 which is 4096, 0x200 which is 512.
    limited_retry: If set to '1', the controller should apply limited retry efforts. If cleared to '0', the controller should apply all
        available error recovery means to return the data to the host.
    force_unit_access: If set to '1', then for data and metadata, if any, associated with logical blocks specified by the Zone Append command, the
        controller shall write that data and metadata, if any, non-volatile media before indicating command completion. There is no implied ordering
        with other commands. If cleared to '0', then this bit has no effect.
    protection_info: Specifies the protection information action and check field, as defined in Figure 355. The Protection Information Check (PRCHK)
        field shall be cleared to 000b.
        Protection Information Action (PRACT): The protection information action bit indicates the action to take for the protection information.
            This bit is only used if the namespace is formatted to use end-to-end protection information.
        Protection Information Check (PRCHK): The protection information check field specifies the fields that shall be checked as part of end-to-end
            data protection processing. This field is only used if the namespace is formatted to use end-to-end protection information.
    protection_info_remap: If this bit is set to '1' then the reference tag in the Protection Information is remapped by the controller as described in
        section 4.4.2. If this bit is cleared to '0', then no remapping of the reference tag is performed by the controller. For Type 1 protection, the
        controller shall abort the command with a status of Invalid Protection Information if this bit is cleared to '0'. For Type 3 protection, the
        controller shall abort the command with a status of Invalid Protection Information if this bit is set to '1'.
    ref_tag: This field indicates the Initial Logical Block Reference Tag value. This field is only used if the namespace is formatted to use
        end-to-end protection information.
    app_tag: This field indicates the Application Tag value. This field is only used if the namespace is formatted to use end-to-end protection
        information.
    app_tag_mask: This field indicates the Application Tag Mask value. This field is only used if the namespace is formatted to use end-to-end
        protection information.
```

Figure 6: Help Info for Zone Append

```
>>> t108.zone_append()         # zone append LBA 0 in zone 0 with default input arguments
Zone append data pattern 0xa5 to starting LBA 0x0 with 0x1 LBAs
0000    A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5         ¥¥¥¥¥¥¥¥¥¥¥¥¥¥¥¥
. . .
Command Completion Queue Status is decoded as follows:
                               CommandSpecific = 0x00000000
                                     Reserved0 = 0x00000000
                               SQ_Head_Pointer = 0x0006
                                 SQ_Identifier = 0x0001
                             Command_Identifier = 0x0704
                    Status_Field:
                                      PhaseBit = 0x01
                                    StatusCode = 0x0000
                                StatusCodeType = 0x00
                                      Reserved = 0x00
                               MoreInformation = 0x00
                                    DoNotRetry = 0x00

>>> t108.zone_management_receive()  # List all zones with default input arguments
Command ZoneManagementReceive passed on port 0 target 108 in tester 192.168.100.111. Output is
decoded as follows:
                                    Num_Zones = 0x0000000000003B98
                Zone_Descriptor_0:
                                      Zone_Type = 0x02
                                     Zone_State = 0x30
```

```
                                Zone_Attributes = 0x00
                                  Zone_Capacity = 0x0000000000003000
                                 Zone_Start_LBA = 0x0000000000000000
                                  Write_Pointer = 0x0000000000000001
                        Zone_Descriptor_1:
                                     Zone_Type = 0x02
                                    Zone_State = 0x10
                                Zone_Attributes = 0x00
                                  Zone_Capacity = 0x0000000000003000
                                 Zone_Start_LBA = 0x0000000000004000
                                  Write_Pointer = 0x0000000000004000
. . .
```

3.2 Multiple Threads I/O in ZNS Examples with SANBlaze Platform

```
>>> help(t107.get_vlun_start_test)
Help on method get_vlun_start_test in module sanblaze_apis:

get_vlun_start_test(test_type='Read', threads=1, blocks=64, ios=0, pass_ios=100, seek_type=0, opcode=12, paused=0, initiator=-1, pattern=2, multipat
h_mode=0, seed=7, dedup=50, comp=1, dup_uniq=50, wblocks=64, wskips=0, skip_blocks=0) method of sanblaze_apis.XML_API instance
    Start a new test to a given port, target and lun.
    test_type = (Read, Write, Compare, Verify, Rewrite)
    threads = number of test threads to run
    blocks = number of blocks per IO (-1 for random)
    ios = number of IOs to do (0 for unlimited)
    pass_ios = for compare tests the number of ios to write before reading back
    seek_type = io access type (0 - Sequential, 1 - Random, 2 - Min/Max, 3 - Butterfly)
    opcode = opcode to use (6,10,12,16, 0 (random))
    paused = start test in paused sate (0,1)
    initiator = initiator test is being run from (-1 for all)
    pattern = pattern being used for writes
        0 - random
        1 - 0x00ff00ff
        2 - 0x55aa55aa
        3 - 8-bit incr
        4 - 8-bit walking 1,0
        5 - 0x0000ffff
        6 - 0x5555aaaa
        7 - 16-bit incr
        8 - 16-bit walking 1,0
        9 - 32-bit incr
        10 - Low Frequency 8B/10B
        11 - Med Frequency 8B/10B
        12 - High Frequency 8B/10B
        13 - Jitter (CJPAT)
        -1 - custom
        -2 - existing data
        (for the following the last to digits can be changed to indicate de-dup percentage). Examples:
        -100 - random, 0% dup
        -101 - random, 1% dup
        -175 - random, 75% dup
        -199 - random, 99% dup
        -300 - random, nodup
        -303 - random, dedup&compression
    multipath_mode = if multiple paths to lun are detected, how to handle them
        0 - default
        1 - this path
        2 - one path
        3 - all paths
        4 - act/opt paths
        5 - act/non-opt paths
    seed = random seed to use
    dedup = dedup ratio
    comp = compression ratio
    dup_uniq = duplicate uniqueness percentage
    wblocks = blocks to write (for Rewrite test)
    wskips = blocks to skip after writing (for Rewrite test)
    skip_blocks = blocks to skip after each IO. Also used for I/O alignment. If alignment is wanted enter the value as a negative value (ex: -4kb)

>>> help(t107.get_vlun_stop_test)
Help on method get_vlun_stop_test in module sanblaze_apis:

get_vlun_stop_test(test_id) method of sanblaze_apis.XML_API instance
    Stop a currently running test of a given port, target, lun and test_id (like "Read_1")
```

Figure 7: Help Info for Start Test and Stop Test

```
>>> t107.zone_management_receive(receive_action=0x10000) # Report zone structure in data buffer
Command ZoneManagementReceive passed on port 0 target 107 in tester 192.168.100.111. Output is
decoded as follows:
                                   Num_Zones = 0x000000000000003F
                        Zone_Descriptor_0:
                                     Zone_Type = 0x02
                                    Zone_State = 0x10
                                Zone_Attributes = 0x00
                                  Zone_Capacity = 0x0000000000018000
                                 Zone_Start_LBA = 0x0000000000000000
```

```
                              Write_Pointer = 0x0000000000000000
                Zone_Descriptor_1:
                                  Zone_Type = 0x02
                                 Zone_State = 0x10
                               Zone_Attributes = 0x00
                                Zone_Capacity = 0x0000000000018000
                               Zone_Start_LBA = 0x0000000000020000
                                Write_Pointer = 0x0000000000020000
. . .

>>>t107.get_vlun_start_test(test_type='Compare',threads=4,blocks=64)#Start 4 threads Compare test
<result>
  <test>
    <status>0</status>
    <test_id>Compare_2</test_id>
    <index>1</index>
  </test>
</result>

>>> t107.zone_management_receive(receive_action=0x10000)  # Report zone structure in data buffer
Command ZoneManagementReceive passed on port 0 target 107 in tester 192.168.100.111. Output is
decoded as follows:
                                  Num_Zones = 0x000000000000003F
                Zone_Descriptor_0:
                                  Zone_Type = 0x02
                                 Zone_State = 0xE0
                               Zone_Attributes = 0x00
                                Zone_Capacity = 0x0000000000018000
                               Zone_Start_LBA = 0x0000000000000000
                                Write_Pointer = 0x00000000FFFFFFFF
                Zone_Descriptor_1:
                                  Zone_Type = 0x02
                                 Zone_State = 0xE0
                               Zone_Attributes = 0x00
                                Zone_Capacity = 0x0000000000018000
                               Zone_Start_LBA = 0x0000000000020000
                                Write_Pointer = 0x00000000FFFFFFFF
. . .

>>> t107.get_vlun_stop_test(test_id='Compare_2')  # Stop the 4 threads Compare test above
<result>
  <status>0</status>
</result>
```

The trace from the SANBlaze platform shows that the 4 threads are running as follows: Each thread is running in one zone, so 4 threads are running in 4 zones. Once complete the first 4 zones begin running on the next 4 zones until they are stopped by user or complete the full test.

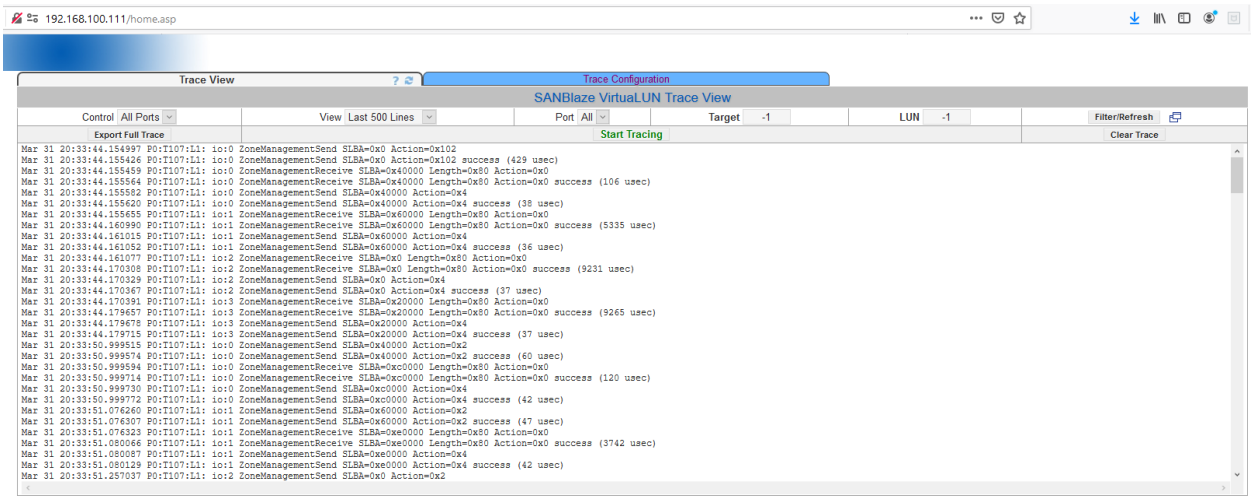

Figure 8: Trace of Multiple Threads I/O in zoned namespace

Summary

In summary, SANBlaze supports all of the Zoned Admin Command Sets and Zoned I/O Command Sets as specified and defined in the latest spec (*NVMe – TP 4053 Zoned Namespaces 2020.03.19 – Final).* SANBlaze provides written scripts that can be run right of the box in our SBExpress GUI, as well as run through our command line interface, XML API interface, and Python wrapped API interface for test automation. SANBlaze is proud to provide a high quality and simple way to test and validate ZNS for your SSD drives.