ZNS를 이용한 리눅스 컨테이너 I/O isolation 기법

팀명: 백견이 불여일타

201724419 김동욱

201724596 채기중

201924445 김지원

지도교수: 안 성 용

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

School of Electrical and Computer Engineering, Computer Engineering Major

Pusan National University

목차

1. 과제 배경 및 목표 3
1.1 과제 배경 3
1.2 기존 문제점 3
1.3 과제 목표 4
2. 요구조건 및 제약사항 분석 5
2.1. 목표 달성을 위한 세부 사항 7
2.2. 제약사항 분석 및 수정사항 7
3. 과제 추진 계획 7
4. 이론적 배경 8
4.1. Zonefs 8
4.2. Dm-zoned Device Mapper 10
4.2.1. Device mapper 10
4.2.2. Dm-zoned overview 11
4.2.3. Dm-zoned metadata 11
4.2.4. dm-zoned 코드 분석 12
5. 중간 결과 22
5.1. 리눅스 컨테이너(LxC) 저장공간 할당 방식 및 cgroup 정보 22
5.2. Zonefs 24
5.3. dm-zoned 25
6. 개발 일정 29
6.1 향후 개발 일정 29

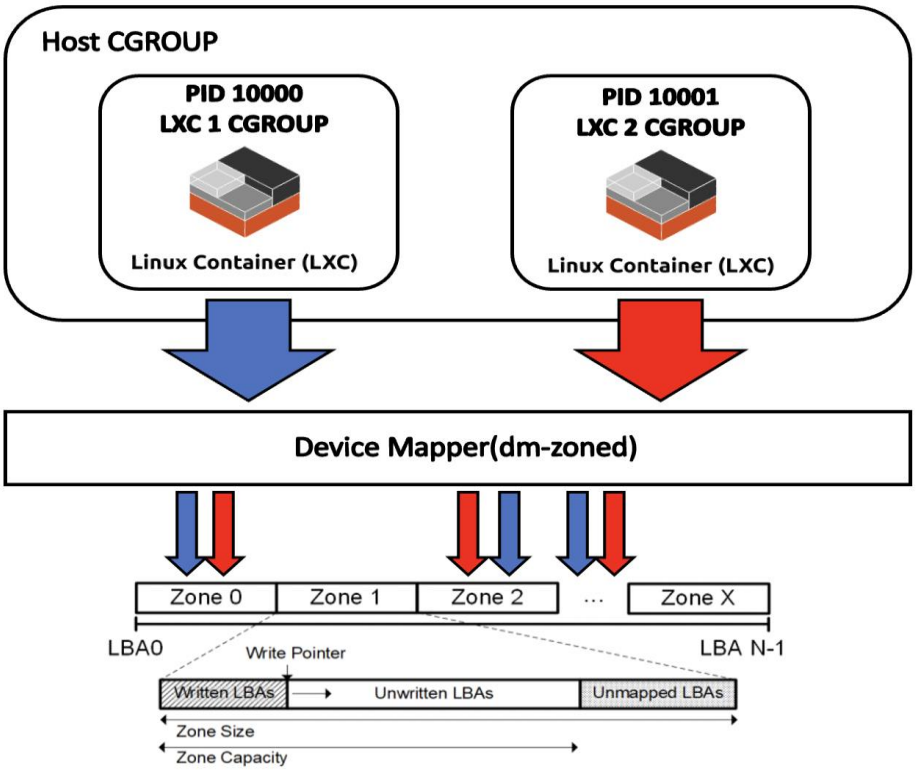
1. 과제 배경 및 목표

1.1 과제 배경

현재 사용되고 있는 일반 SSD는 데이터를 저장할 때 Random Write 방식을 사용한다. 이러한 쓰기 방식은 유효한 데이터와 Garbage 영역의 구분이 없어 효율적인 저장 공간의 사용이 어렵다. 이에 비해 ZNS SSD의 경우 용도와 사용 주기가 동일한 데이터를 Zone에 순차적으로 저장하고 지운다. 이는 서로 다른 I/O Stream 간의 성능 독립을 가능하게 한다.

Linux Container는 런타임 환경에서 애플리케이션을 패키지화하고 분리하는 기술이다. 실행에 필요한 모든 파일(라이브러리, 종속 항목)이 분리되어 동작한다. 따라서 각각의 Zone에 데이터를 나누어 저장하는 ZNS SSD의 특성을 Linux Container에 결합한다면 Container들 사이의 간섭을 최소화하여 성능을 개선할 수 있다고 생각하였다. 우리는 ZNS SSD의 Zone을 각 Container에 독립적으로 분리 할당할 수 있는 정책을 개발하고자 한다.

1.2 기존 문제점

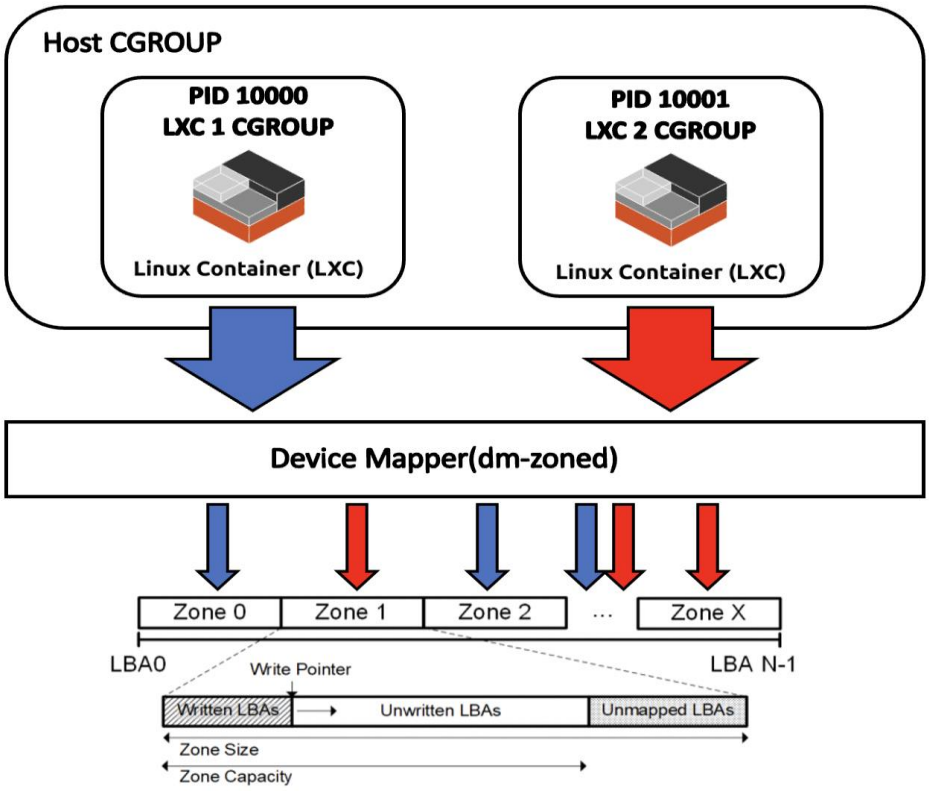


[그림 1] 현재 LxC의 dm-zoned 매핑 방식

현재 zns ssd를 device mapper로 매핑한 후, zns ssd를 컨테이너의 storage pool로 사용해 컨테이너를 생성하게 되면 device mapper를 거쳐서 여러 개의 zone에 컨테이너의 I/O가 발생한다. 컨테이너를 추가로 생성하는 경우 또다시 여러 개의 zone을 할당하게 된다. 이때 비어 있는 zone을 할당하지 않고 그림 1과 같이 기존에 생성된 컨테이너가 사용하고 있는 zone에 append 하는 형식으로 I/O가 발생한다.

즉, 컨테이너마다 독립된 zone을 할당받지 못하고 zone을 공유하므로 zns의 장점을 제대로 가져갈 수 없는 구조이다. Zns ssd가 기존 regular device와 달리 애플리케이션 당 독립된 zone을 사용함으로써 sequential write를 제공한다는 장점이 있는데, 현재 시스템의 구조에서는 각각의 zone에 여러 개의 컨테이너가 할당되어 있으므로 온전한 seq write가 될 수 없다는 문제가 존재한다.

1.3 과제 목표



[그림 2] dm-zoned에서 컨테이너 마다 zone을 할당하는 구조

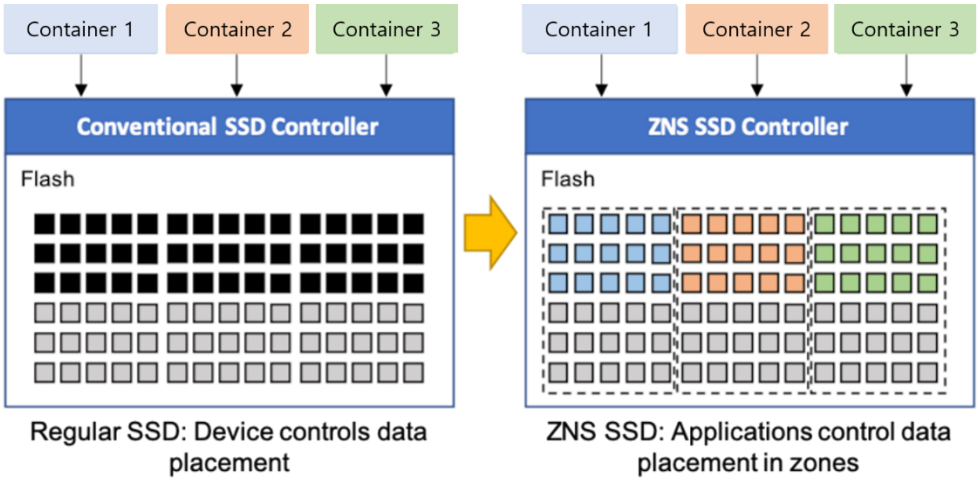
리눅스 컨테이너(LxC)는 PID 단위로 namespace 가 격리된 구조를 가진다는 특징이 있다. 즉 PID 단위로 namespace 가 격리된 구조를 가지므로 서로 다른 컨테이너 각각의 cgroup 을 가지고 있다는 것이다. 현재 컨테이너에 I/O 가 발생하면 dm-zoned 는 cgroup 의 존재를 알지 못한 채 컨테이너마다 zone 을 할당하고 block device 에 I/O 를 진행한다.

그러므로 현재 dm-zoned 에서 zone 을 할당하는 과정에 각각의 컨테이너를 구분할 수 있는 cgroup 을 적용시켜 cgroup 정보에 의해 컨테이너마다 zone 을 할당하도록 하면 컨테이너마다 분리된 zone 을 할당할 수 있고 컨테이너 간 I/O 간섭이 없어질 것이다. 따라서 dm-zoned 에 cgroup 개념을 적용시킨 '컨테이너 zone 분리 할당 알고리즘'을 개발하는 것이 본 팀의 목표이다.

2. 요구조건 및 제약사항 분석

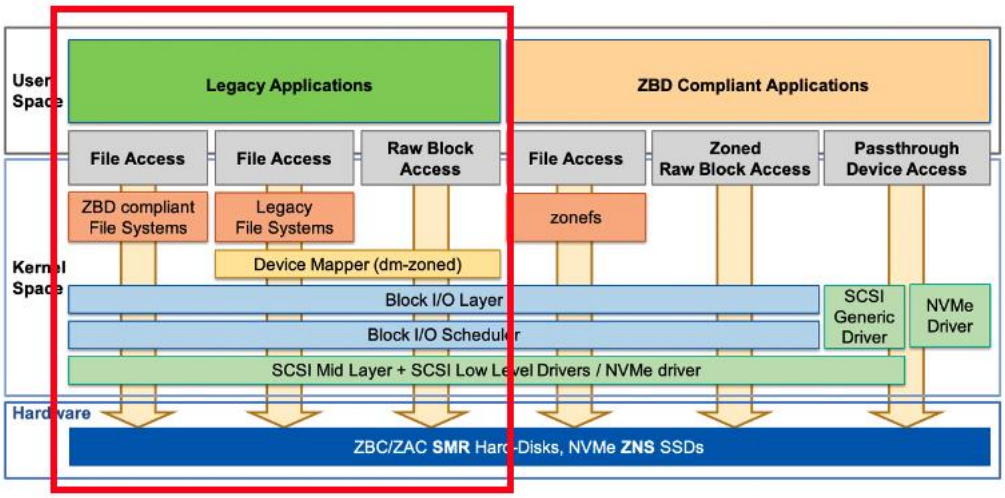
Zoned Namespace 기법을 이용한 저장장치인 ZNS SSD 를 리눅스 컨테이너에 연동하고 device mapper 의 cgroup 에 따른 자원 분배 알고리즘을 개발하여 효율적인 I/O 성능을 얻을 수 있도록 한다.

- 기존의 F2FS, EXT4와 같은 파일 시스템을 사용하는 Legacy application이 Zoned Storage 의 개념을 이해할 수 있도록 한다.
- 각 리눅스 컨테이너는 중첩되지 않는 Zone을 사용함으로써 컨테이너 간 간섭을 최소화 하여 I/O 성능 저하를 줄인다.



[그림 3] 일반 SSD와 ZNS SSD의 저장 방식 차이

그림 3은 기존 SSD와 우리가 목표하는 ZNS SSD의 저장 방식의 차이를 나타낸다. 왼쪽의 그림의 경우, 세 개의 컨테이너에서 들어온 데이터들이 분리되지 않고 섞여서 저장되어 있다. 반면 오른쪽 그림의 경우 각 컨테이너에서 들어온 데이터들이 서로 다른 Zone들을 할당받아 간섭 없이 입출력이 일어나는 것을 확인할 수 있다.



[그림 4] Legacy Application이 ZNS SSD에 I/O하는 과정

그러나 일반적인 Legacy Application이 ZNS SSD에 I/O 명령을 내리기 위해서는 여러 가지 단계를 거쳐야 한다. ZBD를 지원하는 File system을 사용하거나 후술할 Device Mapper를 이용하는 방법이 있는데, 우리는 Dm-zoned Device Mapper를 이용하여 Linux Container가 ZNS SSD에 I/O를 가능케 하도록 한다.

2.1. 목표 달성을 위한 세부 사항

- ZNS SSD를 emulate하여 물리적 저장장치 없이도 Zoned namespace 기법을 사용할 수 있도록 한다.
- 커스텀 커널을 설치하고, 커널 내부에서 실행되는 함수를 추적한 후 수정할 수 있도록 한다.
- 리눅스 컨테이너가 ZNS SSD에 I/O를 실행하도록 수정한다.

2.2. 제약사항 분석 및 수정사항

Host	제약사항	리눅스 커널 코드를 수정 및 해당 코드를 커널에 적용해야 한다.
	해결방안	커스텀 커널을 컴파일 및 설치한다.
Linux Container	제약사항	리눅스 컨테이너가 지원하는 파일 시스템은 directory, zfs, btrfs, lvm 등이고 zonefs는 포함되어 있지 않기 때문에 zonefs를 이용한 storage는 lxc에 사용할 수 없다.
	해결방안	Dm-Zoned Tool을 이용해 Device Mapper Layer를 추가하여 Legacy Application인 리눅스 컨테이너가 ZNS SSD에 I/O를 수행할 수 있도록 한다.
	제약사항	리눅스 컨테이너 내부에서 일어나는 커널 함수 Tracing이 불가능하다.
	해결방안	Sub-cgroup을 생성해 Tracing의 범위를 축소한다.

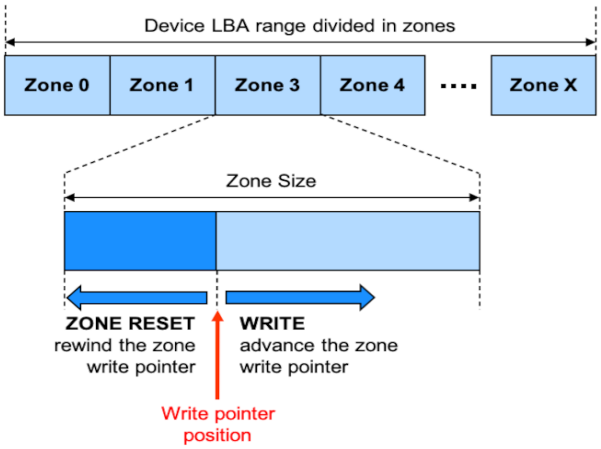
3. 과제 추진 계획

현재 리눅스 컨테이너 내부에서 커널 레벨의 I/O 과정은 일반적인 방법으로는 tracing이 불가능하기 때문에 sub cgroup을 생성해 cgroup을 fio의 옵션으로 설정한다. 이후 fio를 이용해 I/O 과정을 추적하여 dm-zoned에서 자원 분배 과정과 경로를 파악한다. Dm-zoned에서 자원 분배를 하는 함수들을 바탕으로 dm-zoned를 통한 I/O 과정을 분석하여 Zone을 할당하는 알고리즘에 cgroup 개념을 추가하는 방식으로 dm-zoned 코드를 수정한다.

4. 이론적 배경

4.1. Zonefs

Zoned Storage Device(ZSD) 내의 Zone은 순차 쓰기를 필요로 한다. 장치의 주소 공간 내부의 Zone은 다음 쓰기가 수행될 위치를 저장하는 write pointer를 가지며, 데이터는 Zone reset이 일어나기 전에는 직접적으로 덮어씌워질 수 없다.

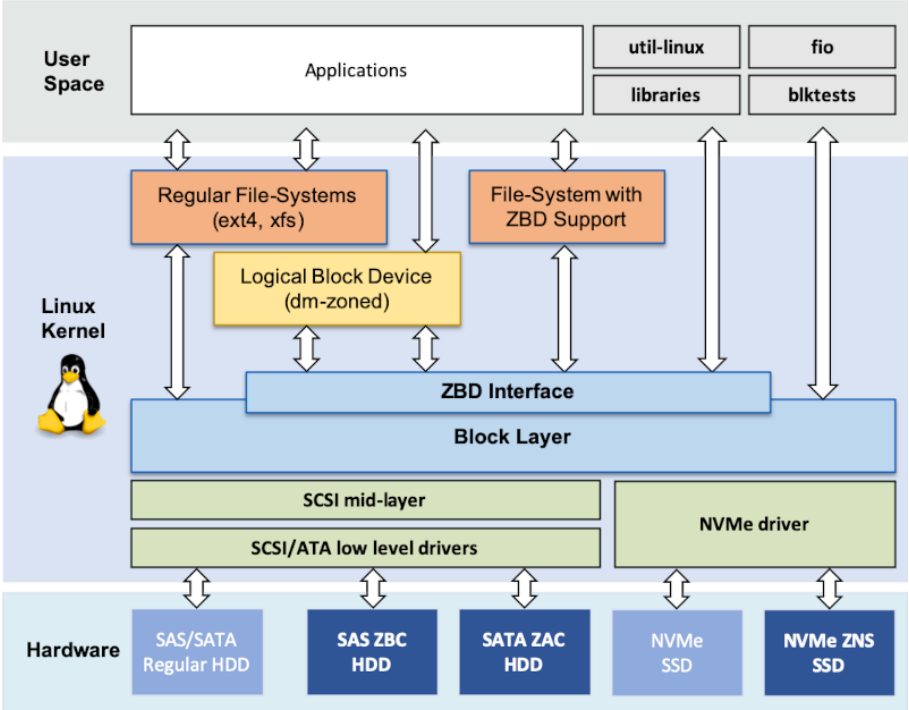


[그림 5] Zoned Storage Devices Principle

Zoned interface는 ZNS SSD에서 사용된다. 하지만 이러한 Zone 기반의 저장 장치들은 기존의 저장 장치들과 다르게 단순히 연결하는 것으로 사용할 수 없다. 이는 위에서 말한 순차 쓰기에 대한 제약 때문인데, 이를 해결하기 위해서는 특정 소프트웨어와 드라이버들의 지원이 필요하다.

리눅스 커널은 4.10.0 버전 이후부터 ZSD에 대한 지원을 시작하였다. Disk driver 단계, File system 단계, Device mapper driver 단계에서 Zoned Storage를 지원하며, 이에 따른 Application 들의 옵션 역시 제공하고 있다. 모든 지원은 Zoned Block Device(ZBD)에 대한 추상화를 기반으로 한다.

ZBD는 Device access protocol과 인터페이스에 독립적이며 ZBD의 추상화는 Zoned storage에 대한 리눅스 커널 지원의 기반이 된다. ZBD 지원은 전통적인 Linux Block Device 인터페이스에 대한 연장이다. ZBD 인터페이스는 Device Driver의 도움을 받아 File System과 같은 커널 서브시스템이 Zone을 관리할 수 있도록 한다.



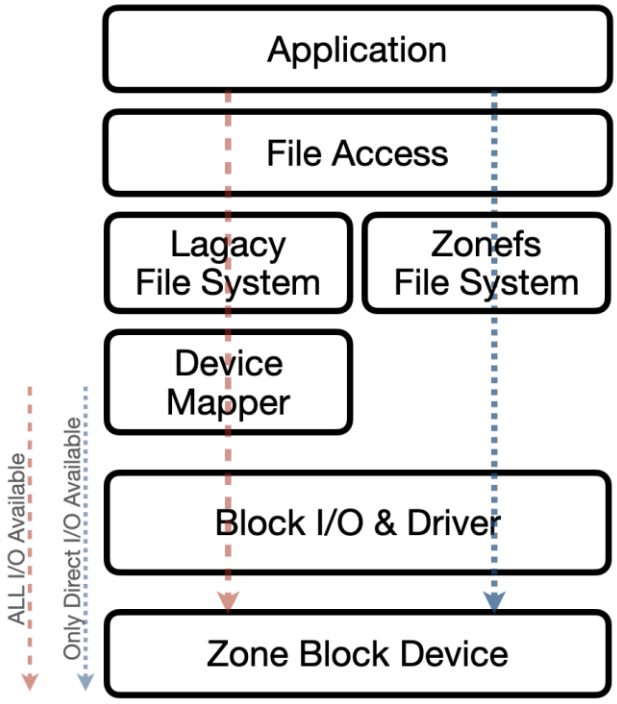
[그림 6] Zoned Storage Device에 대한 지원

그림 6의 Linux kernel 영역 내부에 있는 File-System with ZBD support 중 하나가 zonefs이다. 이는 Zoned Block Device의 각 Zone을 파일 형태로 보여주는 단순한 파일 시스템이다. 일반적인 POSIX 파일 시스템의 ZBD 지원과는 다르게 Zonefs는 순차 쓰기의 제약을 사용자에게 숨기지 않는다. 따라서 Zonefs는 POSIX 파일 시스템에 비해 원시적인 Block Device에 대한 접근을 허용한다. Zone은 Conventional Zone과 Sequential Write Zone으로 나뉘는데, 각 타입은 zonefs에 의해 cnv와 seq라는 디렉토리 하위에서 관리된다.

우리는 Zonefs를 사용하기 위해 mkzonefs라는 도구를 이용하였다. QEMU 상에 Emulate했던 ZNS SSD(nvme0n1)를 포맷하고 이를 zonefs에 마운트 하였다. 이를 통해 파일 시스템의 형태로 zone을 시각화할 수 있었다.

4.2. Dm-zoned Device Mapper

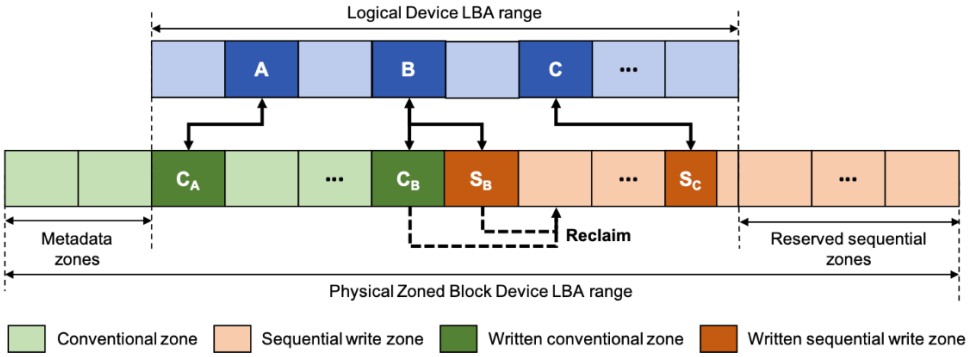
4.2.1. Device mapper



[그림 7] Zone Block Device Application I/O Flow

Zonefs 파일 시스템은 raw block을 Direct I/O를 쉽게 할 수 있도록 추상화된 메서드를 제공하고 zone을 시각화해준다는 장점이 있다. 그러나 zonefs는 기존의 POSIX 파일시스템과 호환되지 않고 Direct I/O만 가능하기 때문에 Linux container와 같은 ZBD Compliant Application이 아닌 일반 Legacy Application에 적용하기 힘들다. 따라서 각 Linux Container마다 다른 zone에 sequential write를 가능하게 하기 위해서는 Device mapper가 필요하다. Dm-zoned Device Mapper는 파일 시스템으로부터 ZBD의 sequential write 제약을 숨김으로써 POSIX 파일 시스템과 ZBD를 이어주는 중간자 역할을 한다.

4.2.2. Dm-zoned overview



[그림 8] Zone mapping overview of the dm-zoned device mapper target

Dm-zoned Device Mapper는 그림 8와 같이 Physical ZBD의 zone 크기의 logical chunk로 나누어진 physical ZBD에 매핑된 Target Device를 만든다. 이러한 Dm zoned Target Device의 모든 zone은 Metadata zones와 Data zones 두 유형으로 나뉜다. Metadata zones는 Data zones에 대한 데이터, 즉 사용자가 저장하는 zone에 대한 정보를 저장하는 zone이고, 이 외 모든 영역에 해당하는 Data zone은 사용자의 데이터 저장 용도로 사용하는 zone이다. Dm-zoned는 random write를 버퍼링 하여 ZBD의 sequential write를 지원한다. 이를 위해 Dm-zoned는 conventional zone 영역이 필요한데 이는 write를 버퍼링하고 sequential zone의 metadata를 random write할 공간을 할당하기 위함이다.

4.2.3. Dm-zoned metadata

- Super block: 첫번째 Conventional zone의 맨 처음 block을 사용하며 Metadata block의 양과 위치 정보를 기록한다.
- Mapping-table blocks: Target Logical Device의 chunk를 Data zone에 매핑하는 테이블 정보와 chunk의 업데이트로 인한 버퍼링에 대한 conventional zone의 정보를 기록한다.
- Bitmap-storage blocks: Data zones의 block의 유효성(삭제와 written의 여부)을 나타내는 비트맵 정보를 기록한다.

Meta data는 conventional zone에 Super block, mapping-table blocks, Bitmap-storage blocks 순으로 구성되어 있다.

4.2.4. dm-zoned 코드 분석

현재 zns ssd에 I/O가 발생할 경우 dm-zoned의 처리 과정을 알아보기 위해 fio와 ftrace를 이용하여 I/O와 연관된 함수를 추적하며 zone이 할당되는 과정을 알아보고 있다. Fio cgroup 옵션을 사용하여 zns ssd를 storage로 사용하는 컨테이너의 cgroup 정보를 추가하여 I/O tracing을 진행하는 과정에서 발견한 코드를 간단히 분석하였다. 현재 fio의 cgroup 정보, jobfile의 I/O size, directory 등을 변경해가며 dm-zoned에서 일어나는 호출에서 확인할 수 있는 구조체와 함수이다.

- dmz_super

Super block의 정보가 들어있는 구조체이다. Mapping table의 개수, bitmap block의 개수 등 Metadata에 대한 정보를 포함하고 있다.

```
struct dmz_super {
        /* Magic number */
        __le32          magic;                  /*   4 */
        /* Metadata version number */
        __le32          version;        /*   8 */
        /* Generation number */
        __le64          gen;                    /*  16 */
        /* This block number */
        __le64          sb_block;       /*  24 */
        /* The number of metadata blocks, including this super block */
        __le32          nr_meta_blocks;         /*  28 */
        /* The number of sequential zones reserved for reclaim */
        __le32          nr_reserved_seq;  /*  32 */
        /* The number of entries in the mapping table */
        __le32          nr_chunks;              /*  36 */
        /* The number of blocks used for the chunk mapping table */
        __le32          nr_map_blocks;          /*  40 */
        /* The number of blocks used for the block bitmaps */
        __le32          nr_bitmap_blocks;  /*  44 */
        /* Checksum */
        __le32          crc;                    /*  48 */
        /* DM-Zoned label */
        u8              dmz_label[32];          /*  80 */
        /* DM-Zoned UUID */
        u8              dmz_uuid[16];           /*  96 */
        /* Device UUID */
        u8              dev_uuid[16];           /* 112 */
        /* Padding to full 512B sector */
        u8              reserved[400];          /* 512 */
};
```

- dmz_metadata

Metadata의 정보가 들어있는 구조체이다. Target을 setup할 때 Metadata 초기화 함수인 dmz_ctr_metadata를 통해 해당 구조체는 초기화되어 metadata 관련 함수에 사용된다.

```
struct dmz_metadata {
        struct dmz_dev              *dev;
        unsigned int                nr_devs;
        char                        devname[BDEVNAME_SIZE];
        char                        label[BDEVNAME_SIZE];
        uuid_t                      uuid;
        sector_t            zone_bitmap_size;
        unsigned int                zone_nr_bitmap_blocks;
        unsigned int                zone_bits_per_mblk;
        sector_t            zone_nr_blocks;
        sector_t            zone_nr_blocks_shift;
        sector_t            zone_nr_sectors;
        sector_t            zone_nr_sectors_shift;
        unsigned int                nr_bitmap_blocks;
        unsigned int                nr_map_blocks;
        unsigned int                nr_zones;
        unsigned int                nr_useable_zones;
        unsigned int                nr_meta_blocks;
        unsigned int                nr_meta_zones;
        unsigned int                nr_data_zones;
        unsigned int                nr_cache_zones;
        unsigned int                nr_rnd_zones;
        unsigned int                nr_reserved_seq;
        unsigned int                nr_chunks;
        /* Zone information array */
        struct xarray               zones;
        struct dmz_sb               sb[2];
        unsigned int                mblk_primary;
        unsigned int                sb_version;
        u64                         sb_gen;
        unsigned int                min_nr_mblks;
        unsigned int                max_nr_mblks;
        atomic_t            nr_mblks;
        struct rw_semaphore         mblk_sem;
        struct mutex                mblk_flush_lock;
        spinlock_t                  mblk_lock;
        struct rb_root              mblk_rbtree;
        struct list_head    mblk_lru_list;
        struct list_head    mblk_dirty_list;
        struct shrinker             mblk_shrinker;
        /* Zone allocation management */
        struct mutex                map_lock;
        struct dmz_mblock   **map_mblk;
        unsigned int                nr_cache;
        atomic_t            unmap_nr_cache;
        struct list_head    unmap_cache_list;
        struct list_head    map_cache_list;
        atomic_t            nr_reserved_seq_zones;
        struct list_head    reserved_seq_zones_list;
        wait_queue_head_t   free_wq;
};
```

- dmz_alloc_mblock

Metadata block에 새로운 block을 할당하는 함수

```
static struct dmz_mblock *dmz_alloc_mblock(struct dmz_metadata *zmd,
                                           sector_t mblk_no)
{
        struct dmz_mblock *mblk = NULL;
        /* See if we can reuse cached blocks */
        if (zmd->max_nr_mblks && atomic_read(&zmd->nr_mblks) > zmd->max_nr_mblks) {
                spin_lock(&zmd->mblk_lock);
                mblk = list_first_entry_or_null(&zmd->mblk_lru_list,
                                                struct dmz_mblock, link);
                if (mblk) {
                        list_del_init(&mblk->link);
                        rb_erase(&mblk->node, &zmd->mblk_rbtree);
                        mblk->no = mblk_no;
                }
                spin_unlock(&zmd->mblk_lock);
                if (mblk)
                        return mblk;
        }
        /* Allocate a new block */
        mblk = kmalloc(sizeof(struct dmz_mblock), GFP_NOIO);
        if (!mblk)
                return NULL;
        mblk->page = alloc_page(GFP_NOIO);
        if (!mblk->page) {
                kfree(mblk);
                return NULL;
        }
        RB_CLEAR_NODE(&mblk->node);
        INIT_LIST_HEAD(&mblk->link);
        mblk->ref = 0;
        mblk->state = 0;
        mblk->no = mblk_no;
        mblk->data = page_address(mblk->page);
        atomic_inc(&zmd->nr_mblks);
        return mblk;
}
```

- dmz_map_zone

Dm-zoned 의 target chunk에 zone을 맵핑한다.

```
/*
 * Map a chunk to a zone.
 * This must be called with the mapping lock held.
 */
void dmz_map_zone(struct dmz_metadata *zmd, struct dm_zone *dzone,
                  unsigned int chunk)
{
        /* Set the chunk mapping */
        dmz_set_chunk_mapping(zmd, chunk, dzone->id,
                              DMZ_MAP_UNMAPPED);
        dzone->chunk = chunk;
```

```
        if (dmz_is_cache(dzone))
                list_add_tail(&dzone->link, &zmd->map_cache_list);
        else if (dmz_is_rnd(dzone))
                list_add_tail(&dzone->link, &dzone->dev->map_rnd_list);
        else
                list_add_tail(&dzone->link, &dzone->dev->map_seq_list);
}
```

- dmz_get_chunk_buffer

Random zone(conventional zone영역)에 할당하고 맵핑하여 이미 sequential zone에 맵핑된 chunk를 버퍼링한다.

```
struct dm_zone *dmz_get_chunk_buffer(struct dmz_metadata *zmd,
                                     struct dm_zone *dzone)
{
        struct dm_zone *bzone;
        int alloc_flags = zmd->nr_cache ? DMZ_ALLOC_CACHE : DMZ_ALLOC_RND;
        dmz_lock_map(zmd);
again:
        bzone = dzone->bzone;
        if (bzone)
                goto out;
        /* Allocate a random zone */
        bzone = dmz_alloc_zone(zmd, 0, alloc_flags);
        if (!bzone) {
                if (dmz_dev_is_dying(zmd)) {
                        bzone = ERR_PTR(-EIO);
                        goto out;
                }
                dmz_wait_for_free_zones(zmd);
                goto again;
        }
        /* Update the chunk mapping */
        dmz_set_chunk_mapping(zmd, dzone->chunk, dzone->id, bzone->id);
        set_bit(DMZ_BUF, &bzone->flags);
        bzone->chunk = dzone->chunk;
        bzone->bzone = dzone;
        dzone->bzone = bzone;
        if (dmz_is_cache(bzone))
                list_add_tail(&bzone->link, &zmd->map_cache_list);
        else
                list_add_tail(&bzone->link, &bzone->dev->map_rnd_list);
out:
        dmz_unlock_map(zmd);
        return bzone;
}
```

- dmz_ctr

Target 를 세팅하는 함수이다. Metadata, device, chunk, bio 를 초기화한다.

```
static int dmz_ctr(struct dm_target *ti, unsigned int argc, char **argv)
{
	struct dmz_target *dmz;
	int ret, i;
	/* Check arguments */
	if (argc < 1) {
		ti->error = "Invalid argument count";
		return -EINVAL;
	}
	/* Allocate and initialize the target descriptor */
	dmz = kzalloc(sizeof(struct dmz_target), GFP_KERNEL);
	if (!dmz) {
		ti->error = "Unable to allocate the zoned target descriptor";
		return -ENOMEM;
	}
	dmz->dev = kcalloc(argc, sizeof(struct dmz_dev), GFP_KERNEL);
	if (!dmz->dev) {
		ti->error = "Unable to allocate the zoned device descriptors";
		kfree(dmz);
		return -ENOMEM;
	}
	dmz->ddev = kcalloc(argc, sizeof(struct dm_dev *), GFP_KERNEL);
	if (!dmz->ddev) {
		ti->error = "Unable to allocate the dm device descriptors";
		ret = -ENOMEM;
		goto err;
	}
	dmz->nr_ddevs = argc;
	ti->private = dmz;
	/* Get the target zoned block device */
	for (i = 0; i < argc; i++) {
		ret = dmz_get_zoned_device(ti, argv[i], i, argc);
		if (ret)
			goto err_dev;
	}
	ret = dmz_fixup_devices(ti);
	if (ret)
		goto err_dev;
	/* Initialize metadata */
	ret = dmz_ctr_metadata(dmz->dev, argc, &dmz->metadata,
				dm_table_device_name(ti->table));
	if (ret) {
		ti->error = "Metadata initialization failed";
		goto err_dev;
	}
	/* Set target (no write same support) */
	ti->max_io_len = dmz_zone_nr_sectors(dmz->metadata);
	ti->num_flush_bios = 1;
	ti->num_discard_bios = 1;
	ti->num_write_zeroes_bios = 1;
	ti->per_io_data_size = sizeof(struct dmz_bioctx);
	ti->flush_supported = true;
	ti->discards_supported = true;
	/* The exposed capacity is the number of chunks that can be mapped */
	ti->len = (sector_t)dmz_nr_chunks(dmz->metadata) <<
		dmz_zone_nr_sectors_shift(dmz->metadata);
	/* Zone BIO */
```

```
        ret = bioset_init(&dmz->bio_set, DMZ_MIN_BIOS, 0, 0);
        if (ret) {
                ti->error = "Create BIO set failed";
                goto err_meta;
        }
        /* Chunk BIO work */
        mutex_init(&dmz->chunk_lock);
        INIT_RADIX_TREE(&dmz->chunk_rxtree, GFP_NOIO);
        dmz->chunk_wq = alloc_workqueue("dmz_cwq_%s",
                                        WQ_MEM_RECLAIM | WQ_UNBOUND, 0,
                                        dmz_metadata_label(dmz->metadata));
        if (!dmz->chunk_wq) {
                ti->error = "Create chunk workqueue failed";
                ret = -ENOMEM;
                goto err_bio;
        }
        /* Flush work */
        spin_lock_init(&dmz->flush_lock);
        bio_list_init(&dmz->flush_list);
        INIT_DELAYED_WORK(&dmz->flush_work, dmz_flush_work);
        dmz->flush_wq = alloc_ordered_workqueue("dmz_fwq_%s", WQ_MEM_RECLAIM,
                                                dmz_metadata_label(dmz->metadata));
        if (!dmz->flush_wq) {
                ti->error = "Create flush workqueue failed";
                ret = -ENOMEM;
                goto err_cwq;
        }
        mod_delayed_work(dmz->flush_wq, &dmz->flush_work, DMZ_FLUSH_PERIOD);
        /* Initialize reclaim */
        for (i = 0; i < dmz->nr_ddevs; i++) {
                ret = dmz_ctr_reclaim(dmz->metadata, &dmz->dev[i].reclaim, i);
                if (ret) {
                        ti->error = "Zone reclaim initialization failed";
                        goto err_fwq;
                }
        }
        DMINFO("(%s): Target device: %llu 512-byte logical sectors (%llu blocks)",
               dmz_metadata_label(dmz->metadata),
               (unsigned long long)ti->len,
               (unsigned long long)dmz_sect2blk(ti->len));
        return 0;
err_fwq:
        destroy_workqueue(dmz->flush_wq);
err_cwq:
        destroy_workqueue(dmz->chunk_wq);
err_bio:
        mutex_destroy(&dmz->chunk_lock);
        bioset_exit(&dmz->bio_set);
err_meta:
        dmz_dtr_metadata(dmz->metadata);
err_dev:
        dmz_put_zoned_device(ti);
err:
        kfree(dmz->dev);
        kfree(dmz);
        return ret;
}
```

- dmz_map

Bio content 를 초기화하고 zone 을 적절하게 나누는 함수이다. 이후 BIO 를 핸들링하기 위해 dmz_queue_chunk_work 함수를 호출한다.

```
static int dmz_map(struct dm_target *ti, struct bio *bio)
{
        struct dmz_target *dmz = ti->private;
        struct dmz_metadata *zmd = dmz->metadata;
        struct dmz_bioctx *bioctx = dm_per_bio_data(bio, sizeof(struct dmz_bioctx));
        sector_t sector = bio->bi_iter.bi_sector;
        unsigned int nr_sectors = bio_sectors(bio);
        sector_t chunk_sector;
        int ret;
        if (dmz_dev_is_dying(zmd))
                return DM_MAPIO_KILL;
        DMDEBUG("(%s): BIO op %d sector %llu + %u => chunk %llu, block %llu, %u blocks",
                dmz_metadata_label(zmd),
                bio_op(bio), (unsigned long long)sector, nr_sectors,
                (unsigned long long)dmz_bio_chunk(zmd, bio),
                (unsigned long long)dmz_chunk_block(zmd, dmz_bio_block(bio)),
                (unsigned int)dmz_bio_blocks(bio));
        if (!nr_sectors && bio_op(bio) != REQ_OP_WRITE)
                return DM_MAPIO_REMAPPED;
        /* The BIO should be block aligned */
        if ((nr_sectors & DMZ_BLOCK_SECTORS_MASK) || (sector & DMZ_BLOCK_SECTORS_MASK))
                return DM_MAPIO_KILL;
        /* Initialize the BIO context */
        bioctx->dev = NULL;
        bioctx->zone = NULL;
        bioctx->bio = bio;
        refcount_set(&bioctx->ref, 1);
        /* Set the BIO pending in the flush list */
        if (!nr_sectors && bio_op(bio) == REQ_OP_WRITE) {
                spin_lock(&dmz->flush_lock);
                bio_list_add(&dmz->flush_list, bio);
                spin_unlock(&dmz->flush_lock);
                mod_delayed_work(dmz->flush_wq, &dmz->flush_work, 0);
                return DM_MAPIO_SUBMITTED;
        }
        /* Split zone BIOs to fit entirely into a zone */
        chunk_sector = sector & (dmz_zone_nr_sectors(zmd) - 1);
        if (chunk_sector + nr_sectors > dmz_zone_nr_sectors(zmd))
                dm_accept_partial_bio(bio, dmz_zone_nr_sectors(zmd) - chunk_sector);
        /* Now ready to handle this BIO */
        ret = dmz_queue_chunk_work(dmz, bio);
        if (ret) {
                DMDEBUG("(%s): BIO op %d, can't process chunk %llu, err %i",
                        dmz_metadata_label(zmd),
                        bio_op(bio), (u64)dmz_bio_chunk(zmd, bio),
                        ret);
                return DM_MAPIO_REQUEUE;
        }
        return DM_MAPIO_SUBMITTED;
}
```

- dmz_queue_chunk_work

Bio를 핸들링 하기 전, bio chunk work를 받아오는 함수이며 dmz_map 함수에 의해 호출된다. Bio 리스트에 chunk work의 bio를 추가한다.

```
static int dmz_queue_chunk_work(struct dmz_target *dmz, struct bio *bio)
{
	unsigned int chunk = dmz_bio_chunk(dmz->metadata, bio);
	struct dm_chunk_work *cw;
	int ret = 0;
	mutex_lock(&dmz->chunk_lock);

	cw = radix_tree_lookup(&dmz->chunk_rxtree, chunk);
	if (cw) {
		dmz_get_chunk_work(cw);
	} else {

		cw = kmalloc(sizeof(struct dm_chunk_work), GFP_NOIO);
		if (unlikely(!cw)) {
			ret = -ENOMEM;
			goto out;
		}
		INIT_WORK(&cw->work, dmz_chunk_work);
		refcount_set(&cw->refcount, 1);
		cw->target = dmz;
		cw->chunk = chunk;
		bio_list_init(&cw->bio_list);
		ret = radix_tree_insert(&dmz->chunk_rxtree, chunk, cw);
		if (unlikely(ret)) {
			kfree(cw);
			goto out;
		}
	}
	bio_list_add(&cw->bio_list, bio);
	if (queue_work(dmz->chunk_wq, &cw->work))
		dmz_get_chunk_work(cw);
out:
	mutex_unlock(&dmz->chunk_lock);
	return ret;
}
```

- dmz_chunk_work

Dm-zoned가 처리해야 하는 work를 전달받아 I/O를 하도록 하는 함수이다. Chunk work의 target을 dmz의 target으로 설정한 후 처리해야 할 bio를 bio_list_pop을 이용해 가져온다. 이후 dmz_handle_bio 함수에 dmz, cw, bio를 인자로 받는 dmz handler 함수를 호출한다. Work 처리가 정상적으로 이루어지면 dmz_put_chunk_work함수를 호출해 완료한 work를 전달하여 처리해야 할 work의 count를 감소시킨다.

```
static void dmz_chunk_work(struct work_struct *work)
{
	struct dm_chunk_work *cw = container_of(work, struct dm_chunk_work, work);
	struct dmz_target *dmz = cw->target;
	struct bio *bio;
	mutex_lock(&dmz->chunk_lock);
	/* Process the chunk BIOs */
	while ((bio = bio_list_pop(&cw->bio_list))) {
		mutex_unlock(&dmz->chunk_lock);
		dmz_handle_bio(dmz, cw, bio);
		mutex_lock(&dmz->chunk_lock);
		dmz_put_chunk_work(cw);
	}
	/* Queueing the work incremented the work refcount */
	dmz_put_chunk_work(cw);
	mutex_unlock(&dmz->chunk_lock);
}
```

- dmz_handle_bio

dmz_chunk_work에서 전달받은 dmz, cw, bio를 가지고 I/O 핸들러를 매핑하는 함수이다. Chunk에 매핑된 zone 정보를 가져와서 write를 하는 경우 zone을 active 상태로 변경한다. Bio의 I/O 타입에 따라 적절한 핸들러를 호출한다. 예를 들어 write인 경우 dmz_handle_write를 호출한다.

```
static void dmz_handle_bio(struct dmz_target *dmz, struct dm_chunk_work *cw,
			   struct bio *bio)
{
	struct dmz_bioctx *bioctx =
		dm_per_bio_data(bio, sizeof(struct dmz_bioctx));
	struct dmz_metadata *zmd = dmz->metadata;
	struct dm_zone *zone;
	int ret;
	dmz_lock_metadata(zmd);

	zone = dmz_get_chunk_mapping(zmd, dmz_bio_chunk(zmd, bio),
				     bio_op(bio));
	if (IS_ERR(zone)) {
		ret = PTR_ERR(zone);
		goto out;
	}
	/* Process the BIO */
	if (zone) {
```

```
                dmz_activate_zone(zone);
                bioctx->zone = zone;
                dmz_reclaim_bio_acc(zone->dev->reclaim);
        }
        switch (bio_op(bio)) {
        case REQ_OP_READ:
                ret = dmz_handle_read(dmz, zone, bio);
                break;
        case REQ_OP_WRITE:
                ret = dmz_handle_write(dmz, zone, bio);
                break;
        case REQ_OP_DISCARD:
        case REQ_OP_WRITE_ZEROES:
                ret = dmz_handle_discard(dmz, zone, bio);
                break;
        default:
                DMERR("(%s): Unsupported BIO operation 0x%x",
                      dmz_metadata_label(dmz->metadata), bio_op(bio));
                ret = -EIO;
        }

        if (zone)
                dmz_put_chunk_mapping(zmd, zone);
out:
        dmz_bio_endio(bio, errno_to_blk_status(ret));
        dmz_unlock_metadata(zmd);
}
```

- dmz_bio_endio

Dm-zoned 에서 block I/O 가 종료되면 호출되는 함수이다. Bioctx 구조체는 zone 정보를 가지고 있다. Write 작업을 완료한 경우 zone 이 할당되어 있으므로 해당 zone 이 정상적으로 I/O 가 완료되었으면 active 상태인 zone 을 deactivate 상태로 바꾼다. I/O 가 정상적으로 완료되지 못한 경우 zone 을 error 상태로 변경한다.

```
static inline void dmz_bio_endio(struct bio *bio, blk_status_t status)
{
        struct dmz_bioctx *bioctx =
                dm_per_bio_data(bio, sizeof(struct dmz_bioctx));
        if (status != BLK_STS_OK && bio->bi_status == BLK_STS_OK)
                bio->bi_status = status;
        if (bioctx->dev && bio->bi_status != BLK_STS_OK)
                bioctx->dev->flags |= DMZ_CHECK_BDEV;
        if (refcount_dec_and_test(&bioctx->ref)) {
                struct dm_zone *zone = bioctx->zone;
                if (zone) {
                        if (bio->bi_status != BLK_STS_OK &&
                            bio_op(bio) == REQ_OP_WRITE &&
                            dmz_is_seq(zone))
                                set_bit(DMZ_SEQ_WRITE_ERR, &zone->flags);
                        dmz_deactivate_zone(zone);
                }
                bio_endio(bio);
        }
}
```

5. 중간 결과

5.1. 리눅스 컨테이너(LxC) 저장공간 할당 방식 및 cgroup 정보

컨테이너가 생성되는 과정에서 컨테이너의 저장 공간은 Host OS에 연결된 storage pool에 저장된다. 현재 컨테이너의 storage pool은 Host OS의 /(Root)에 매핑되어있다. 그림 9는 컨테이너가 생성되기 전 Host OS 파일 시스템의 디스크 정보이다. 현재 Root 디렉토리의 사용 중인 공간은 13G이다.

```
kijung@kijung-qemu-server:~$ df -h
Filesystem                         Size  Used Avail Use% Mounted on
udev                                62G     0   62G   0% /dev
tmpfs                               13G  1.2M   13G   1% /run
/dev/mapper/ubuntu--vg-ubuntu--lv   23G   13G  8.7G  60% /
tmpfs                               62G     0   62G   0% /dev/shm
tmpfs                              5.0M     0  5.0M   0% /run/lock
tmpfs                               62G     0   62G   0% /sys/fs/cgroup
/dev/loop0                          62M   62M     0 100% /snap/core20/1328
/dev/loop1                          62M   62M     0 100% /snap/core20/1518
/dev/loop2                          68M   68M     0 100% /snap/lxd/21835
/dev/loop3                          44M   44M     0 100% /snap/snapd/14978
/dev/loop4                          68M   68M     0 100% /snap/lxd/22753
/dev/loop5                          47M   47M     0 100% /snap/snapd/16010
/dev/sda2                          1.5G  217M  1.2G  16% /boot
tmpfs                               13G     0   13G   0% /run/user/1000
tmpfs                              1.0M     0  1.0M   0% /var/snap/lxd/common/ns
```

[그림 9] 컨테이너 생성 전 Host OS 파일시스템 디스크 정보

그림 10은 Host OS에서 컨테이너가 생성된 후 파일 시스템의 디스크 정보이다. Root 디렉토리의 사용중인 공간은 14G로 1G가 증가한 반면 그림 9에서 확인할 수 있는 컨테이너의 총 저장공간은 4G이다. 이 사실을 바탕으로 컨테이너가 저장공간을 사용할 때 Host OS가 동적으로 저장공간을 할당하는 것을 알 수 있다.

```
kijung@kijung-qemu-server:~$ df -h
Filesystem                         Size  Used Avail Use% Mounted on
udev                                62G     0   62G   0% /dev
tmpfs                               13G  1.2M   13G   1% /run
/dev/mapper/ubuntu--vg-ubuntu--lv   23G   14G  8.1G  63% /
tmpfs                               62G     0   62G   0% /dev/shm
tmpfs                              5.0M     0  5.0M   0% /run/lock
tmpfs                               62G     0   62G   0% /sys/fs/cgroup
/dev/loop0                          62M   62M     0 100% /snap/core20/1328
/dev/loop1                          62M   62M     0 100% /snap/core20/1518
/dev/loop2                          68M   68M     0 100% /snap/lxd/21835
/dev/loop3                          44M   44M     0 100% /snap/snapd/14978
/dev/loop4                          68M   68M     0 100% /snap/lxd/22753
/dev/loop5                          47M   47M     0 100% /snap/snapd/16010
/dev/sda2                          1.5G  217M  1.2G  16% /boot
tmpfs                               13G     0   13G   0% /run/user/1000
tmpfs                              1.0M     0  1.0M   0% /var/snap/lxd/common/ns
```

[그림 10] 컨테이너 생성 후 Host OS 파일시스템 디스크 정보

```
root@kijungContainer1:~# df -h
Filesystem                          Size  Used Avail Use% Mounted on
default/containers/kijungContainer1  4.4G  420M  4.0G  10% /
none                                492K  4.0K  488K   1% /dev
udev                                 62G     0   62G   0% /dev/tty
tmpfs                               100K     0  100K   0% /dev/lxd
tmpfs                               100K     0  100K   0% /dev/.lxd-mounts
tmpfs                                62G     0   62G   0% /dev/shm
tmpfs                                62G  8.2M   62G   1% /run
tmpfs                               5.0M     0  5.0M   0% /run/lock
tmpfs                                62G     0   62G   0% /sys/fs/cgroup
```

[그림 11] 생성된 컨테이너의 파일시스템 디스크 정보

그림 12은 컨테이너가 생성될 때 컨테이너의 cgroup이 함께 생성되는 것을 보여준다. 즉, 컨테이너가 생성될 때 cgroup sub directory가 함께 만들어지고 컨테이너는 해당 cgroup 정보를 바탕으로 동작한다.

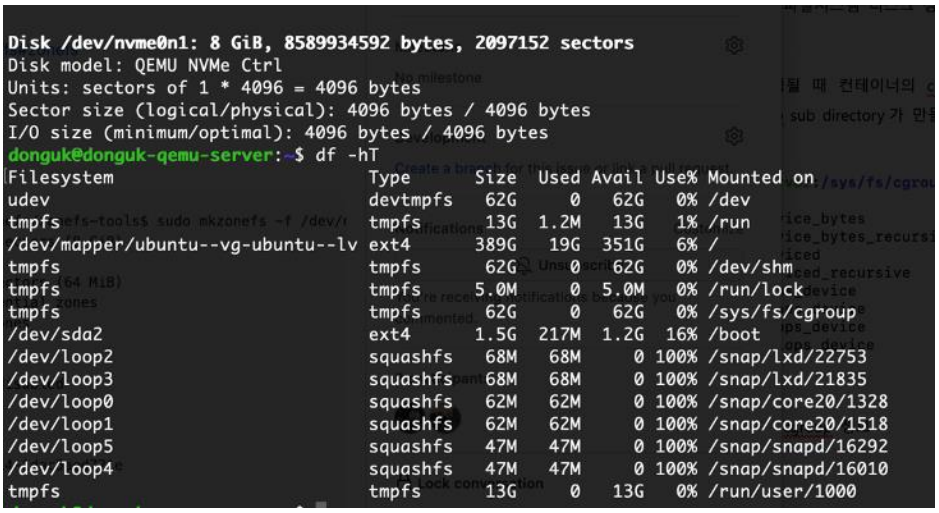
```
kijung@kijung-qemu-server:/sys/fs/cgroup/blkio$ ls
blkio.reset_stats                       cgroup.procs
blkio.throttle.io_service_bytes         cgroup.sane_behavior
blkio.throttle.io_service_bytes_recursive  init.scope
blkio.throttle.io_serviced              lxc.monitor.kijungContainer1
blkio.throttle.io_serviced_recursive    lxc.payload.kijungContainer1
blkio.throttle.read_bps_device          notify_on_release
blkio.throttle.read_iops_device         release_agent
blkio.throttle.write_bps_device         system.slice
blkio.throttle.write_iops_device        tasks
cgroup.clone_children                   user.slice
```

[그림 12] 생성된 컨테이너의 cgroup 정보

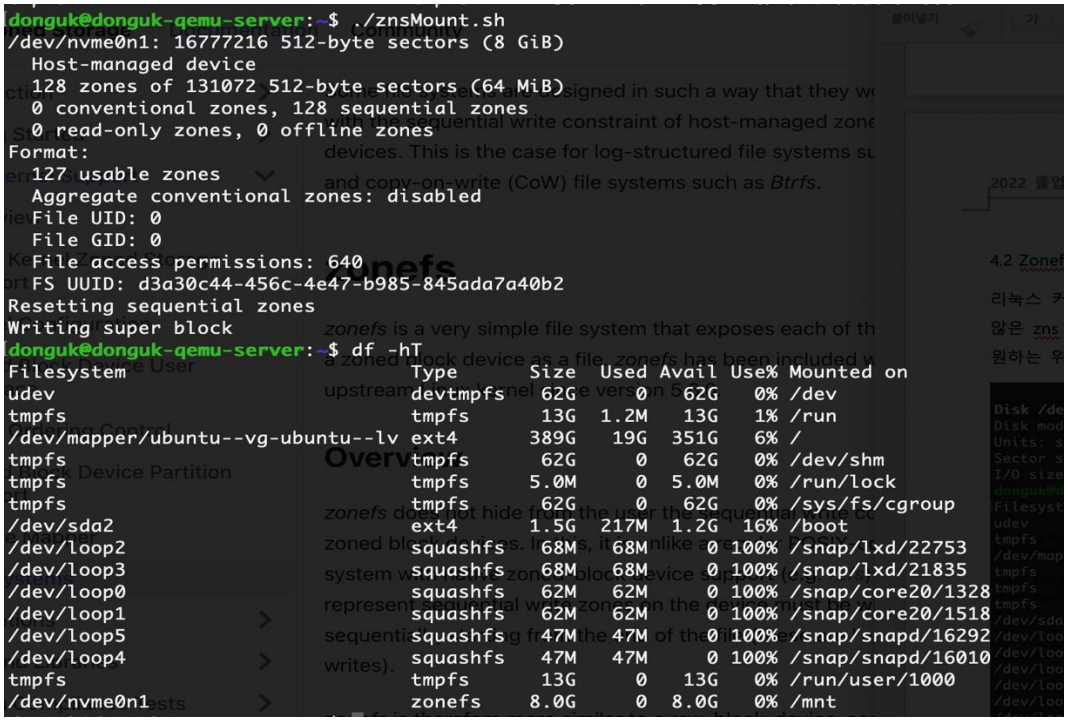
5.2. Zonefs

리눅스 커널은 5.6.0버전부터 zonefs file system을 지원한다. 현재 Qemu 서버에는 마운트 되지 않은 zns ssd /dev/nvme0n1 디스크가 존재한다. 이 디스크를 zonefs file system으로 설정하고 원하는 위치에 마운트 하게 되면 zonefs를 지원하는 디스크로 사용할 수 있다.

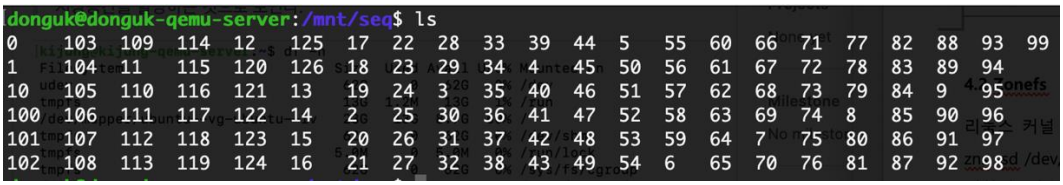
그림 13, 14는 /dev/nvme0n1 디스크를 zonefs file system을 이용해 마운트 한 모습이다. 그림 15와 같이 마운트 된 위치로 이동하여 디렉토리를 조회해 보면 seq디렉토리와 seq 디렉토리 내부에서 zone number를 확인할 수 있다. 4.1에서 컨테이너가 Host OS의 디렉토리를 storage pool로 사용할 수 있음을 보았고, ZNS SSD의 마운트 된 위치인 /mnt 디렉토리를 컨테이너의 storage pool로 사용하려고 시도하였으나 lxc에서 지원하는 컨테이너 storage pool의 file system type은 Directory, Btrfs, LVM, ZFS, Ceph RBD, CephFS이기 때문에 zonefs 기반의 storage pool을 사용할 수 없다. 따라서 legacy application 기반으로 설계를 변경하였다.



[그림 13] ZNS SSD를 zonefs를 이용해 마운트한 모습



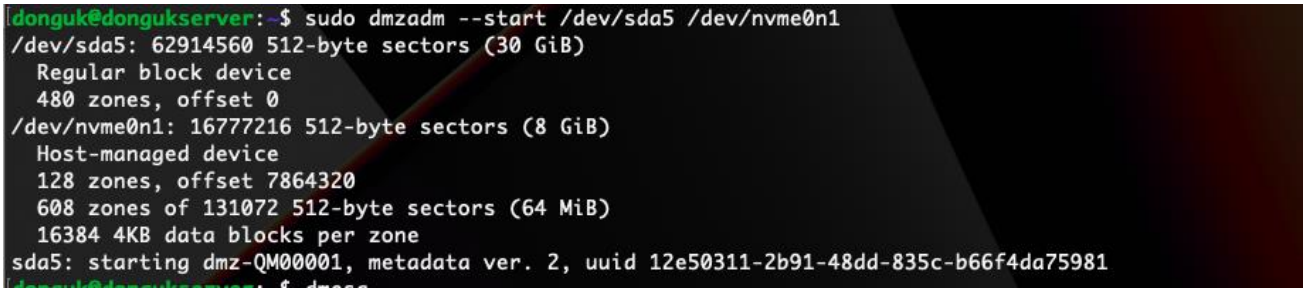
[그림 14] 마운트 후의 zns ssd



[그림 15] zns ssd의 zone number

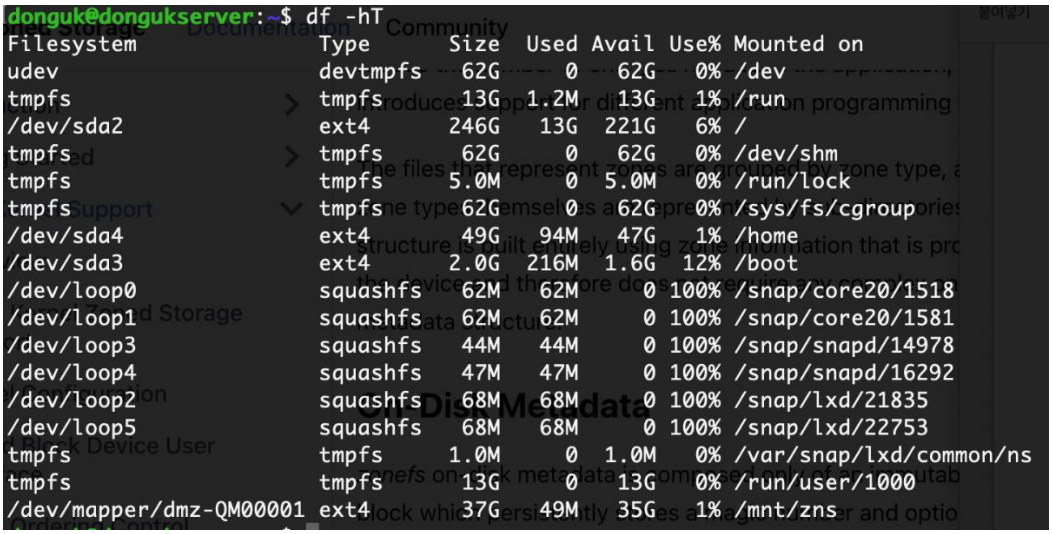
5.3. dm-zoned

Linux Container는 Zonefs 파일시스템을 지원하지 않는 Legacy 타입의 Application이기 때문에device mapper를 이용해 ZNS SSD에 접근하는 방식을 사용하기로 했다. Zone을 지원하는 device mapper는 dm-zoned이다. ZNS NVMe SSD는 conventional zone을 지원하지 않기 때문에 conventional zone의 역할을 하는 regular block storage가 필요하다. 따라서 Dm-zoned의 target에 conventional zone의 역할을 하는 regular block storage를 추가하여 ZNS SSD를 매핑하였다.



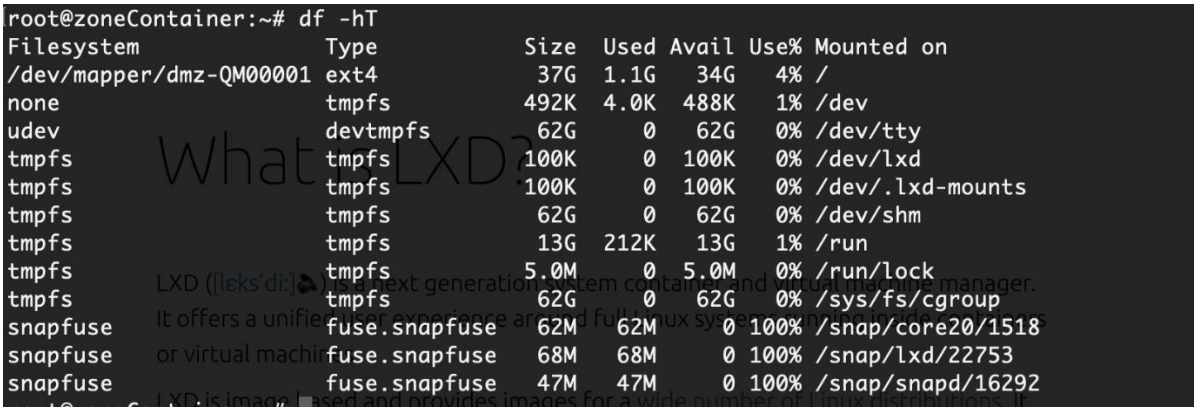
[그림 16] Dm-zoned를 이용한 ZNS SSD매핑

그림 16은 Dm-zoned Tool을 이용해 regular block인 /dev/sda5와 ZNS SSD인 /dev/nvme0n1을 Target에 매핑한 것이다. 이후 /dev/sda5의 파일시스템을 ext4로 설정하고 원하는 위치에 마운트 한다. 그림 17에서 /mnt 위치에 정상적으로 마운트 된 것을 확인할 수 있다.



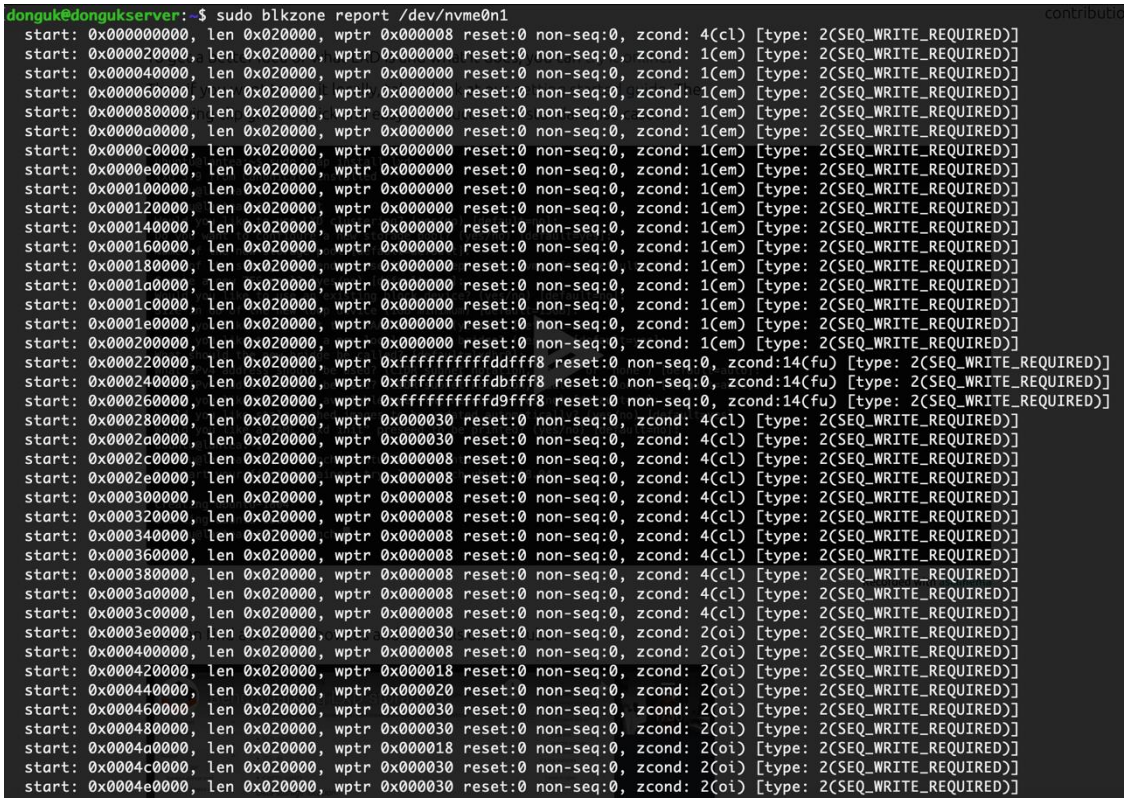
[그림 17] device mapper로 매핑된 storage를 마운트한 후의 디스크 정보

Device mapper Target을 컨테이너에서 지원하는 ext4 file system으로 설정하였기 때문에 컨테이너의 storage pool로 설정할 수 있다. 그림 18을 보면 컨테이너가 logical block device인 dm-zoned을 컨테이너 storage로 사용하는 것을 알 수 있다. 즉 컨테이너가 device mapper를 거쳐 ZNS SSD를 저장장치로 사용한다는 것이다.



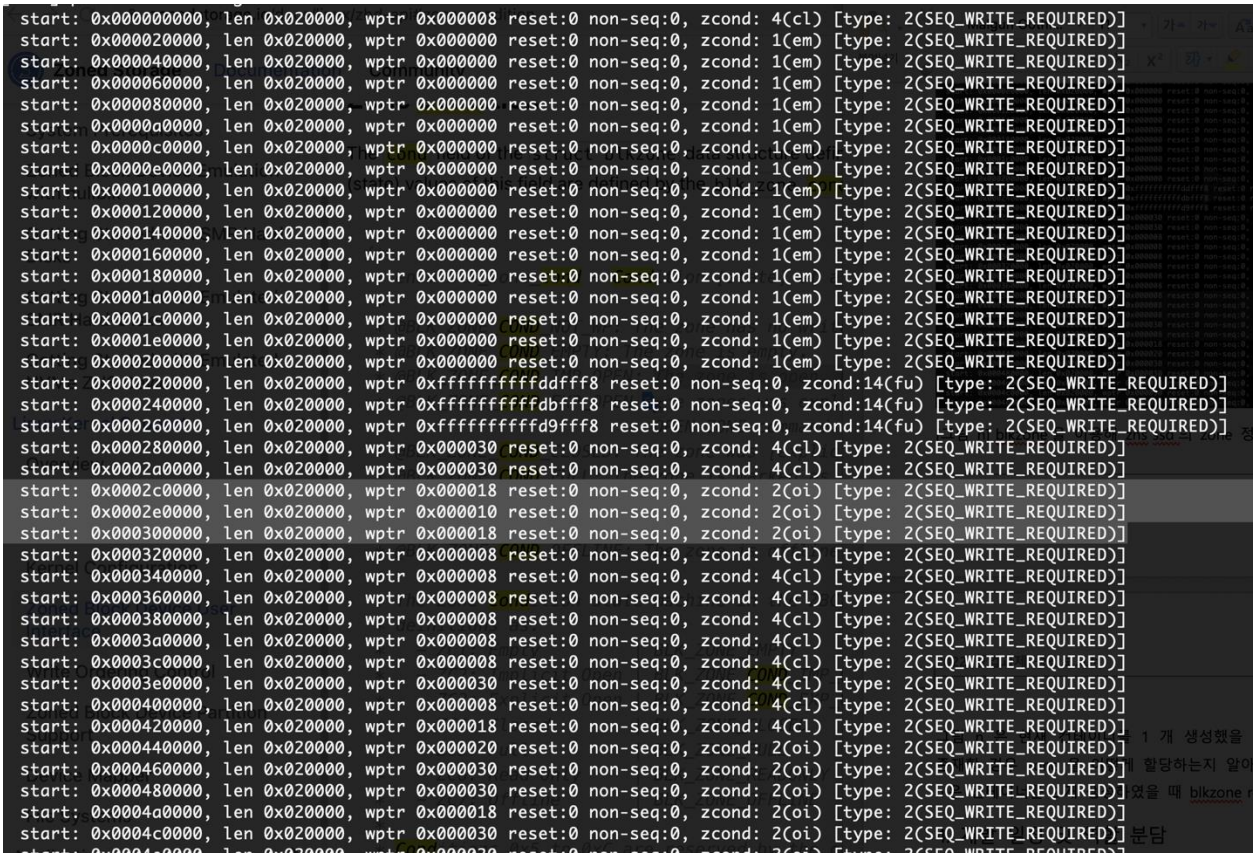
[그림 18] dm-zoned device mapper를 컨테이너의 storage로 사용

ZNS SSD에 컨테이너 내부 파일들이 정상적으로 write 되었는지의 여부는 blkzone report로 확인할 수 있다. 그림 19는 zns ssd /dev/nvme0n1의 zone 정보를 보여준다. Zcond의 값을 통해 특정 zone의 상태를 확인할 수 있는데, 계속해서 write 할 수 있는 zone은 oi(open), write가 정상적으로 종료된 zone은 cl(close), zone 용량을 모두 사용한 zone은 fu(full)로 표현된다.



[그림 19] blkzone을 이용한 ZNS SSD의 zone 정보 확인

그림 19는 컨테이너를 1개 생성했을 때의 blkzone report 결과이다. 컨테이너가 여러 개 존재할 경우 zone을 어떻게 할당하는지 알아보기 위해 컨테이너를 1개 더 생성하였다. 그림 20은 컨테이너를 2개 생성하였을 때의 blkzone report 결과이다. 컨테이너를 추가로 생성했을 때 기존 생성되어 있던 컨테이너가 사용하는 zone을 함께 사용하는 것을 알 수 있다. 이는 0x0002c0000의 시작 주소를 가진 zone의 wptr(write pointer)가 이동했다는 것으로 알 수 있다. 즉, 현재의 컨테이너들은 분리된 zone을 할당받지 못하고 기존에 사용하던 zone에 append 하는 식으로 동작한다.



[그림 20] 컨테이너를 2개 생성했을 때의 blkzone report 결과

6. 개발 일정

6.1 향후 개발 일정

월 / 항목	7				8				9				
주차	1	2	3	4	1	2	3	4	1	2	3	4	5
중간 보고서 작성													
컨테이너 별 zone 분리 할당 정책 개발													
성능 비교 및 분석 1													
디버깅													
성능 비교 및 분석 2													
최종 보고서작성													

Reference

1. https://zonedstorage.io/
2. https://www.kernel.org/doc/html/latest/filesystems/zonefs.html
3. https://github.com/westerndigitalcorporation/zonefs-tools
4. https://linuxcontainers.org/lxd/introduction/
5. https://manpages.ubuntu.com/
6. https://access.redhat.com/
7. https://blackinkgj.github.io/