Posted by : Jaeeun Kim at Jun 25, 2019

볼륨은 어떻게 관리될까?

- 볼륨헤더(Volume Header)와 섹터테이블(Sector Table) -

앞선 포스터 (Overview)에서 볼륨 매니저가 섹터의 예약(reservation)을 관리한다고 이야기하였다. 이번 글에서는 볼륨내의 섹터들이 어떻게 관리되는지에 대한 구체적인 이야기와 이를 위해 볼륨이 어떻게 구성되어 있는지를 다룬다. 여기서 다루어지는 볼륨의 구조는 그대로 non-volatile memory (SSD, HDD 등)에 쓰여진다.

볼륨 구조

디스크매니저의 가장 큰 역할은 파일생성과 확장을 위해 섹터들을 제공해주는 것이다. 이를 위해 각 볼륨은 파일들에게 할당해줄 섹터들과 이를 관리하기 위한 메타(meta)데이터로 이루어져있다. 메타데이터들이 저장된 페이지를 볼륨의 **시스템페이지(System Page)**라고 하며, 볼륨에 대한 정보와 각 섹터들의 예약여부를 담고 있다. 시스템페이지는 다음과 같이 두가지로 나뉜다.

- **볼륨 헤더 페이지 (Volume Header Page, 이하 헤더페이지)**: 페이지 크기, 볼륨내 섹터의 전체/최대 섹터, 볼륨 이름 등, 볼륨에 대한 정보를 지니고 있는 페이지
- **섹터 테이블 페이지 (Sector Table Page, 이하 STAB페이지)**: 볼륨내의 각 섹터의 예약여부를 비트맵으로 들고 있는 페이지

이러한 시스템페이지들은 볼륨이 생성될 때 미리 볼륨 내의 정해진 공간에 쓰여지고, 이 페이지들이 포함된 섹터를 제외한 나머지 섹터들이 파일매니저로부터의 섹터예약요청을 처리하기위해 사용된다. 볼륨헤더는 볼륨의 첫번째 한 개의 페이지에 할당되고, STAB 페이지는 헤더페이지의 바로 다음 페이지부터 볼륨의 크기를 모두 커버할 수 있는 만큼의 양이 연속적으로 할당받는다 (*disk_stab_init()*). 이를 도식화 하면 다음과 같다.

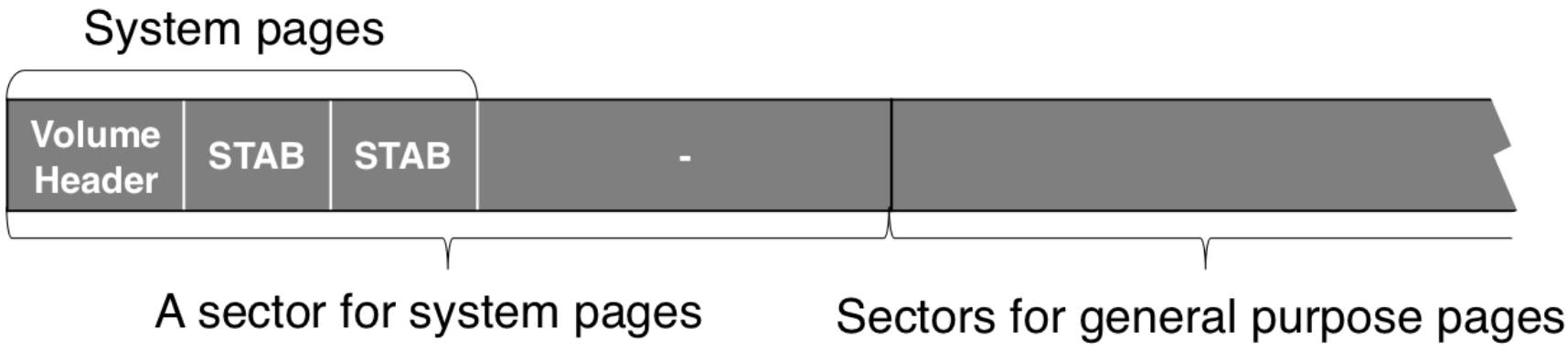

Figure 1: Volume Format

첫 섹터가 시스템페이지들을 위해 할당되어 있는 모습을 볼 수 있다. 볼륨의 시스템페이지들의 수가 한 섹터를 못 채울 경우 그림처럼 시스템페이지들을 위해 할당된 섹터 내의 페이지들이 일부 사용되지 않을 수 있고, 볼륨에 크기가 커지면 이에 따라 시스템페이지들을 위한 섹터가 둘 이상 할당될 수도 있다.

볼륨 헤더 (Volume Header)

볼륨헤더(*DISK_VOLUME_HEADER*)는 볼륨의 첫번째 페이지에 쓰여지며, 기본적으로 볼륨에 대한 정보들이 고정사이즈로 들어가고 나머지공간에는 가변길이 변수들이 들어간다. 볼륨 헤더가 담고 있는 정보는 크게 5가지 정도로 분류할 수 있다.

- 볼륨 정보: 볼륨 자체에 대한 정보로 볼륨전체에 공통적으로 적용되는 정보이다. 볼륨의 타입, 캐릭터셋, 생성시간, 섹터당 페이지수, 페이지의 크기 등이 저장된다.
- 섹터 정보: 볼륨의 현재 섹터의 정보이다. 볼륨내에 몇개의 섹터가 있는지, 얼마나 확장될 수 있는지 등이 저장된다.
- 시스템페이지 정보: 앞서 이야기한 시스템페이지에 대한 정보들이 저장된다.
- 체크포인트 정보: 마지막으로 체크포인트가 성공시 체크포인트의 시작 지점의 로그 레코드 LSA 정보가 저장된다. 이는 리커버리과정에서 사용된다.
- 가변길이 변수: 볼륨헤더페이지내에서 볼륨헤더의 모든 고정변수들을 제외한나머지 공간은 가변길이 변수들을 위한 공간이다. 볼륨의 full path나 사용자 정의 comment등이 저장된다.
- 기타: reserved 등 동작과 무관한 특수목적 변수들이 저장된다.

구체적으로 볼륨 헤더 구조체(*DISK_VOLUME_HEADER*)가 담고 있는 정보(변수)들은 다음과 같다.

분류	변수 타입	변수명	설명
볼륨	INT8	db_charset	데이터베이스의 캐릭터셋
	INT16	volid	해당 볼륨의 볼륨 식별자
	DB_VOLTYPE	type	볼륨의 타입, 볼륨이 어떻게 관리될지를 결정 Permanent: 영구적으로 볼륨유지 Temporary: 서버 종료/재시작시 제거. 임시데이터를 저장하는데 기존 볼륨의 공간이 부족할 경우 생성된다.
	DB_VOLPURPOSE	purpose	볼륨의 이용목적, 볼륨을 어떻게 사용할지를 결정 Permanent: 영구적인 데이터를 저장할 것. Temporary: 임시적인 데이터를 저장할 것. 임시데이터를 저장할 때에 임시타입의 볼륨을 만들기전에 임시목적의 영구타입볼륨이 있을 경우 먼저 사용한다.
	INT64	db_creation	데이터베이스 생성시간
	INT16	next_volid	여러 볼륨이 있을 경우 그들을 연결하는 포인터, 다음 볼륨의 식별자를 담음
	DKNPAGES	sect_npgs	한 섹터당 페이지 수
	INT16	iopagesize	한 페이지의 크기
	HFID	boot_hfid	볼륨 부팅과 멀티 볼륨관련된 정보를 담고있는 힙(Heap)파일의 식별자
섹터	DKNPAGES	nsect_total	볼륨의 현재 총 섹터 수, 볼륨파일의 크기를 결정
	DKNPAGES	nsect_max	볼륨이 확장될 수 있는 최대 크기의 섹터 수
	SECTID	hint_allocsect	섹터예약시 섹터테이블의 어디부터 탐색할지 캐싱해둔 값
시스템페이지	DKNPAGES	stab_npages	섹터테이블이 차지하는 페이지 수
	PAGEID	stab_first_page	섹터테이블의 시작페이지
	PAGEID	sys_lastpage	마지막 시스템 페이지 (현재 stab_first_page+stab_npages -1)
체크포인트	LOG_LSA	chkpt_lsa	체크포인트 시작점의 LSA, 리커버리분석의 시작점 (ARIES의 master record)
가변길이변수	char [1]	var_fields	가변길이 변수들의 시작점, var_fileds + offset_to_* 가 각 가변변수의 위치
	INT16	offset_to_vol_fullname	볼륨의 절대경로 이름의 offset
	INT16	offset_to_next_vol_fullname	next_volid 볼륨의 절대경로 이름의 offset
	INT16	offset_to_vol_remarks	볼륨에 대한 코멘트의 offset 코멘트는 볼륨포맷(*disk_format()*)시에 적히는 것으로 유저가 addvoldb를 실행하면서 적는 코멘트나 볼륨의 공간이 가득차 자동으로 새로운 볼륨을 만들어질 경우 적히는 코멘트 ("Automatic Volume Extension") 등이 들어간다.

기타	INT32	reserved0/1/2/3	미래 확장성을 위한 예약변수들
	INT8/32	dummy1/2	alignment를 위한 더미변수들
	char []	magic	볼륨파일의 매직넘버

> 각 변수들에 대한 설명을 달아두었긴 했지만 명확한 이해를 위해서는 각 변수들의 값들이 언제 설정되고, 어떻게 사용되는지 등을 알아야한다. 이에 대한 자세한 내용은 각 변수들이 이용되는 부분을 설명할 때에 다시 살펴보도록 한다.

섹터 테이블 (Sector Table)

섹터테이블(STAB)은 볼륨내 모든 섹터들의 사용여부(예약여부)를 저장하고 있는 비트맵이다. 섹터테이블페이지의 하나의 비트는 하나의 섹터의 예약 여부를 나타낸다. 섹터테이블은 볼륨헤더페이지의 바로 다음페이지(볼륨의 두번째 페이지, *stab_first_page*)부터 시작하여 볼륨의 최대크기(*nsect_max*)를 커버할 수 있는 만큼의 페이지(*stab_npages*)를 사용한다.

> 현재 코드는 볼륨헤더페이지의 다음페이지가 *stab_first_page*, STAB의 마지막 페이지가 *sys_lastpage* 와 같은 값을 가지고 있지만 데이터의 구조상 또 다른 시스템페이지가 추가될 수 있을 것으로 보인다.

섹터예약에 관한 연산을 수행할 때, 각 비트들을 하나씩 순회하며 연산을 수행할 수도 있지만 큐브리드는 비트들을 ***DISK_STAB_UNIT* (이하 unit, 유닛)**이라는 단위로 묶어 관리, 연산하고 불가피할 때에만 비트를 순회한다. 비트연산을 할 때에 CPU 아키텍쳐등을 고려하여 효율적인 방법으로 처리 할 수 있도록 이러한 처리단위를 제공하는 것으로 보인다. 정리하자면 섹터테이블의 비트맵은 여러페이지로 구성되며 각 페이지는 다시 유닛으로 나뉘고, 유닛의 비트들은 각각의 하나의 섹터의 예역여부를 나타낸다. 섹터테이블을 읽거나 조작하는 등의 연산은 모두 이 유닛을 기반으로 이루어진다.

> 현재 유닛은 다음과 같이 UINT64형이다. CPU아키텍처나 디자인에 맞춰 이 값을 변경시키면 STAB의 관리 단위를 변경 시킬 수 있다. 주석 또한 이 값의 변경을 통해 유닛단위를 쉽게 변경할 수 있을 것이라 이야기하고 있다.
>
> ```
> typedef UINT64 DISK_STAB_UNIT;
> // ... If we ever want to change the type of unit, this can be modified and should be handled automatically, ...
> ```
>
> 하지만, 유닛을 단위로 하는 연산들이 *bit64_count_zeros()* 등으로 64bit에 맞게 하드코딩되어 있기에 변경이 그렇게 단순해 보이지는 않는다. 각 바이트단위로 비트연산 함수들은 base/bit.c에 모두 구현되어 있기 때문에 유닛 사이즈에 맞춰 각 비트연산을 호출하게끔 수정한다면 코드의 유연성을 확보할 수 있을 것으로 보인다.

만약 sector_id가 32100인 섹터에 대한 예약여부를 확인하려할 때, STAB에서 해당 비트의 위치는 어떻게 구할 수 있을까? 이는 마치 초에서 (시,분,초)를 구하듯 (page_id, offset_to_unit, offset_to_bit) 으로 다음과 같이 계산된다.

```
page_id: (볼륨헤더의 stab_first_page) + sector_id / (페이지의 비트 수)
offset_to_unit: sector_id % (페이지의 비트 수) / (페이지내 유닛의 수)
offset_to_bit: sector_id % (페이지의 비트 수) % (페이지내 유닛의 수)
```

만약 1KB페이지, 64bit unit이라면 sector_id 32100인 (3, 117, 36)이 된다.

안타깝게도 페이지의 크기가 2^n형태가 아니기 때문에 OS의 페이지테이블이나 CPU 캐시처럼 단순 비트 쉬프트연산으로 유닛과 오프셋등을 구할 수 없다. 때문에 비싼 /, % 연산이 사용된다.

> IO 페이지의 크기는 4KB, 16KB 등 2^n형태이더라도 모든 페이지가 공통적으로 페이지타입, LOG_LSA 등의 공간을 이미 예약해두었기 때문에 실제 사용가능한 크기는 이 영역을 제외한 크기이다.

섹터 테이블의 연산

섹터의 예약정보를 조회하거나 예약하려면 섹터테이블의 비트맵을 조작해야한다. 이러한 연산들은 앞서말한 유닛단위를 기반으로 이루어지며, 하나의 섹터비트나 유닛을 참조할 일 보다는 여러 유닛들을 참조하는 경우가 대부분이기 때문에 **커서(Cursor, *DISK_STAB_CURSOR*)**와 이터레이션 인터페이스(*disk_stab_iterate_units()*)를 제공한다. 커서는 볼륨내 한 섹터의 STAB에서의 위치(page_id, offset_to_unit, offset_to_bit)를 가리킨다. 또, 커서가 가리키는 유닛에 대한 연산을 위해 커서가 가리키고 있는 유닛의 포인터(page, unit)를 들고 있다.

```
typedef struct disk_stab_cursor DISK_STAB_CURSOR;
struct disk_stab_cursor
{
    const DISK_VOLUME_HEADER *volheader;    /* Volume header */

    PAGEID pageid;      /* Current page ID */
    int offset_to_unit;     /* Offset to current unit in page. */
    int offset_to_bit;      /* Offset to current bit in unit. */

    SECTID sectid;      /* Sector ID */

                // 위의 변수들은 모두 현재 커서가 가리키는 섹터에 대한 정보와 STAB내에서 섹터의 위치
                // 아래의 변수들은 위의 변수들이 가리키는 STAB내의 유닛을 참조하기 위한 포인터

    PAGE_PTR page;      /* Fixed table page. */
    DISK_STAB_UNIT *unit;       /* Unit pointer in current page. */
};
```

이터레이션 함수인 *disk_stab_iterate_units()* 의 선언부는 다음과 같다. (설명에 필요하지 않은 인자들은 제외하였다.)

```
static int disk_stab_iterate_units (..., DISK_STAB_CURSOR * start, DISK_STAB_CURSOR * end,
DISK_STAB_UNIT_FUNC f_unit, void *f_unit_args)
```

앞서 이야기한 커서 자료형의 *start, end*와 이터레이션하면서 유닛에 적용할 함수(*DISK_STAB_UNIT_FUNC*)와 함수의 인자를 매개변수로 받는 것을 볼 수있다. 이 함수는 [*start, end*) 범위의 유닛을 순회하면서 각 유닛마다 *DISK_STAB_UNIT_FUNC*함수를 적용시킨다. 여타 함수형 프로그래밍언어에 있는 map()함수를 생각하면 이해가 쉽다. *start, end* 커서는 *disk_stab_cursor_set_at_*()* 류의 함수를 통해 STAB의 시작이나 끝, 특정 sector ID로 설정된다. *DISK_STAB_UNIT_FUNC* 는 함수포인터로 다음과 같다.

```
typedef int (*DISK_STAB_UNIT_FUNC) (..., DISK_STAB_CURSOR * cursor, bool * stop, void *args);
```

*disk_stab_iterate_units()*에서 이터레이션되어 만나는 각 유닛에 대한 커서를 인자로 받아 사용자가 정의한 작업을 진행한다. 이 때 *stop*에 true를 넣고 함수를 종료하면, *disk_stab_iterate_units()* 의 이터레이션이 종료된다. 예를 들어 30개의 섹터를 예약하려 할 때, 이번 유닛에서 30개의 섹터 예약을 모두 완료했다면 더 이상의 작업을 중지하는 종료조건으로 활용할 수 있다. 이러한 유닛이터레이션을 통한 연산에는 섹터들 예약, 섹터들 예약 해제, 가용섹터들의 갯수확인등이 있다. 좀 더 확실한 이해를 위해 가용섹터들의 갯수확인에 사용되는 *DISK_STAB_UNIT_FUNC* 는 *disk_stab_count_free()* 와 이에 대한 호출부를 살펴보자.

```
// free sector의 갯수를 구하는 함수 정의
static int disk_stab_count_free (THREAD_ENTRY * thread_p, DISK_STAB_CURSOR * cursor, bool * stop, void
*args)
{
    DKNSECTS *nfreep = (DKNSECTS *) args;

    /* add zero bit count to free sectors total count */
    *nfreep += bit64_count_zeros (*cursor->unit);
    return NO_ERROR;
}

// 함수 호출부
int disk_rv_volhead_extend_redo (THREAD_ENTRY * thread_p, LOG_RCV * rcv)
{
        ...
        disk_stab_cursor_set_at_sectid (volheader, volheader->nsect_total - nsect_extend,
&start_cursor);
        disk_stab_cursor_set_at_end (volheader, &end_cursor);
                error_code = disk_stab_iterate_units (thread_p, volheader, PGBUF_LATCH_READ,
&start_cursor, &end_cursor, disk_stab_count_free, &nfree);
        ...
    disk_cache_update_vol_free (volheader->volid, nfree);
        ...
}
```

호출부의 예는 recovery의 redo phase에 사용되는 함수중 하나인 *disk_rv_volhead_extend_redo()* 로, 실제로 확장된 볼륨내의 free setor의 갯수를 디스크 캐시에 업데이트하기 위한 코드이다. 확장하기 전의 위치(*volheader->nsect_total - nsect_extend*)에 *start*커서를 두고, stab의 끝에 *end*커서를를 두고 *disk_stab_iterate_units()*함수를 호출하여 [*start, end*)를 순회하며 모든 유닛들에서 0인 비트들의 갯수를 구하는 것을 볼 수 있다.

> 이러한 이터레이션 방식은 파일매니저와 디스크매니저의 여러 곳에서 사용된다. 대표적으로 나중에 살펴볼 파일 매니저의 파일테이블과 유저테이블등에서도 이러한 패턴으로 데이터를 접근, 조작한다.

이어서 다룰 디스크매니저 내용은 다음과 같다.

1. 섹터 예약 및 예약해제
2. 볼륨 확장