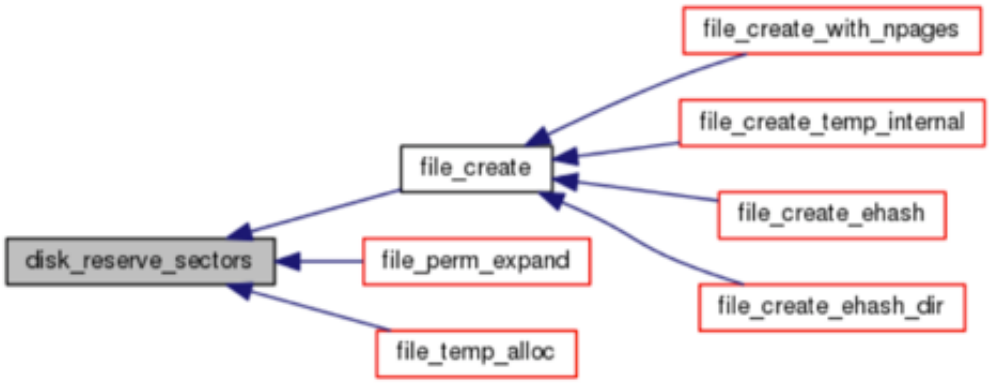
섹터 예약은 어떻게 이루어질까?

- 2 단계 섹터 예약 (2-Step Sector Reservation) -

볼륨 내의 파일이 이전에 예약한 섹터를 모두 사용하여 추가적인 페이지를 할당할 수 없다면 디스크 매니저에게 추가적인 섹터를 요청한다. 디스크 매니저는 이러한 섹터 예약요청을 어떻게 처리할까? 이전 글 의 내용으로 예상할 수 있듯이, *disk_stab_iterate_units()*함수를 통해 볼륨 파일의 섹터테이블을 순회하며 가용한 섹터가 있는지 확인하고 해당 비트를 set 해줌으로써 섹터를 예약했음을 표시할 수 있을 것이다. 기본적으로 섹터 예약이란 이처럼 디스크에 있는 섹터테이블을 변경시키는 것은 맞지만, 큐브리드는 이에 앞서 한 단계 과정을 더 수행한다. 이번 글에서는 섹터 예약이 어떻게 이루어지는 자세히 알아본다.

> 언제 섹터 예약 요청이 발생할까?
>
> 섹터 예약함수(*disk_reserve_sectors()*)를 호출하는 경우를 살펴보면 아래의 그래프에서 볼수 있듯이 새로운 파일이 생성될 때, 혹은 파일에서의 페이지할당과정에서 이전에 예약해둔 섹터를 모두 사용하여 파일을 확장해야 할 때이다.
>
> 
>

Sector Reservation Overview

큐브리드의 섹터 예약은 다음과 같은 **두 단계**로 이루어진다.

1. **디스크 캐시 (Disk Cache)를 통해 섹터를 사전 예약**
    1. 현재 볼륨들로 섹터예약이 가능한지 확인, 불가능할 경우 공간 확보 (볼륨 확장, 추가)
    2. 섹터를 예약할 볼륨들 선택, 각 볼륨별로 몇개의 섹터를 예약할지 결정
2. **사전예약에서 결정된 사항대로 섹터테이블을 순회하면서 예약비트 세팅**

디스크캐시는 영구/임시볼륨의 개수, 볼륨별 전체 섹터 수, 가용 섹터 수등 전체 볼륨에 대한 섹터들의 정보를 가지고 있다. 즉, 섹터테이블과 볼륨헤더의 정보를 바탕으로 예약 시 사용되는 정보들을 미리 계산하여 메인메모리에 저장해 둔다. 실제로 예약을 하기 전에 디스크캐시를 이용한 사전 예약을 함으로써 다음과 같은 문제들을 해결한다.

1. 현재 볼륨들에 요청된 섹터 예약을 처리할 만큼의 충분한 공간이 있는지 확인하기 위해 각 볼륨의 섹터테이블을 순회해야 한다.

2. 어떤 볼륨이 가용 섹터를 가지고 있어 예약을 처리해줄 수 있는지 확인하기 위해 각 볼륨의 섹터테이블을 순회해야 한다.
3. 섹터테이블 순회를 통해 예약공간을 찾았으나, 트랜잭션 A가 섹터를 예약하는 중에 트랜잭션 B가 예약을 해버려서 공간이 부족해지고, 실패할 수 있다(Concurrency).

큐브리드는 섹터의 예약 가능 여부를 판단할 때 직접 섹터테이블을 탐색하지 않고 미리 계산된 디스크 캐시를 통하여 이 여부를 판단한다. 덕분에 1, 2의 비효율적인 문제를 해결할 수 있다. 3번의 동시성문제 또한 디스크캐시의 정보를 바탕으로 사전예약을 함으로써 단순 값 비교의 짧은 뮤텍스(mutex)로 해결할 수 있다. 사전예약은 공간이 부족한 것을 확인하면 OS에게 추가적인 공간을 요청하는 등의 실제 예약시 필요한 모든 작업을 수행한다. 덕분에 실제 예약에서는 섹터테이블을 순회하여 정해진만큼의 비트를 set 하는 것만으로 안전하게 예약을 수행할 수 있다.

2단계 섹터 예약과정을 함수레벨로 좀 더 자세히 살펴보면 다음과 같다. 아래에서 이 과정을 자세히 설명하겠지만, 많은 함수가 사용되고 이름도 유사하여 먼저 전체적인 흐름과 함께 살펴보고 가면 이해가 더 쉬울 것이다. 들여쓰기는 함수의 호출을 뜻하며 핵심적인 함수 외에는 제외하였다.

```
disk_reserve_sectors ()
  disk_reserve_from_cache () // step 1: pre-reservation with Disk Cache
    disk_reserve_from_cache_vols () // conditional, 기존 볼륨의 가용 섹터만으로 예약이 가능할 경우
      disk_reserve_from_cache_volume()
    disk_extend () // conditional, 기존 볼륨의 가용 섹터만으로 예약이 불가능할 경우
      disk_volume_expand ()
      disk_reserve_from_cache_volume ()
      disk_add_volume () // conditional, 확장한 볼륨만으로는 예약이 불가능할 경우
      disk_reserve_from_cache_volume ()
  disk_reserve_sectors_in_volume () // step 2: actual reservation with Sector Table
```

- ***disk_reserve_sectors()***: 임의의 볼륨들에서 n개의 섹터 예약을 요청
- ***disk_reserve_from_cache()***: step 1 - 디스크 캐시를 조회하고 변경하여 사전예약을 수행한다. 이때, 사전예약을 위한 뮤텍스를 잡는다.
- ***disk_reserve_from_cache_vols()***: 요청된 n개의 섹터를 예약하기 위해 디스크 캐시의 볼륨들을 순회하며 각 볼륨에 사전예약을 요청한다(*disk_reserve_from_cache_volume()*).
- ***disk_reserve_from_cache_volume()***: 특정 볼륨에 요청받은 섹터수만큼 사전예약을 시도한다. 이때 요청량이 해당 볼륨의 가용 섹터보다 많다면 가용 섹터만큼만 사전예약을 수행한다. 사전예약 수행이란 디스크캐시에 있는 정보를 변경하는 것을 말한다.
- ***disk_extend()***: 디스크캐시를 참고하여 가용 섹터가 부족하다면 사전예약 전에 먼저 추가적인 공간을 확보한다. 이때 공간을 확보(확장, 추가)하면 확장한 볼륨에서 사전예약(*disk_reserve_from_cache_volume()*)을 시도한다.
- ***disk_volume_expand()***: 추가적인 공간확보방법 중 하나로 먼저 볼륨을 확장한다. 이때 확장하는 볼륨은 최근 추가한 마지막 볼륨이다. 한 볼륨을 최대크기로 확장하고 나서야 새로운 볼륨을 추가하므로, 확장가능한 볼륨은 항상 하나이다.
- ***disk_add_volume()***: 볼륨을 최대 크기로 확장했음에도 가용 섹터가 부족하다면 새로운 볼륨을 추가한다.
- ***disk_reserve_sectors_in_volume()***: step 2 - 실제적인 예약표시과정으로 step 1에서 약속된 내용으로 단순히 섹터테이블을 수정한다(이전 글).

Sector Reservation in details

섹터 예약과정을 자세히 살펴보기에 앞서 과정 전체에서 사용되는 몇 가지 구조체를 살펴보자.

DISK_RESERVE_CONTEXT

큐브리드는 예약과정 동안 요청진행 상황을 저장/참고하기 위하여 *DISK_RESERVE_CONTEXT* 구조체를 사용한다. 이 구조체는 예약과정 시작 시에 요청정보를 바탕으로 만들어져 예약이 끝날 때까지의 진행과정을 저장한다. 어떤 변수들을 저장하고 있는지 확인해보면 예약과정을 어느 정도 가늠해볼 수 있다. 구조체가 지닌 변수를 살펴보면 다음과 같다.

변수	설명
int nsect_total	예약 요청된 섹터수
VSID *vsid	섹터(sectid, volid)의 배열. 예약과정의 최종산출물로 예약한 섹터들의 위치이다. 섹터 요청시 배열을 할당해 넣어주면, 예약된 섹터들을 채워준다.
DISK_CACHE_VOL_RESERVE cache_vol_reserve[VOLID_MAX]	볼륨별 사전예약 섹터 수(VOLID, DKNSECTS)의 배열. Step 1에서 사전예약결과로 어떤 볼륨에서 얼마나 예약할지를 저장한다. Step 2에서 이정보를 바탕으로 볼륨을 찾아 섹터테이블을 변경한다.
int n_cache_vol_reserve	사전예약한 섹터들이 속한 볼륨의 수. cache_vol_reserve 배열의 크기.
int n_cache_reserve_remaining	아직 사전예약처리되지 못한 섹터 수. 예약 시작시 nsect_total로 초기화되고 사전예약과정에서 하나씩 감소한다. 0이되면 사전예약이 끝난다.
DKNSECTS nsects_lastvol_remaining	Step 2를 수행할 때 현재 예약중인 볼륨에서의 남은 섹터 예약량을 저장. cache_vol_reserve[volid]의 사전예약량을 초기값으로하여 섹터테이블을 변경해가며 감소시킨다.
DB_VOLPURPOSE purpose	예약 목적 (temporary or permanent).

DISK_CACHE

디스크 캐시는 앞서 말했던 것처럼 전체 볼륨들의 섹터 예약정보들을 바탕으로 예약 시 필요한 값들을 미리 계산하여 가지고 있다. 이 값들을 조회/변경함으로써 섹터테이블 변경 전에 사전예약을 수행한다. 디스크 캐시는 *DISK_CACHE* 구조체로 표현되며 *disk_Cache*라는 전역변수로 접근된다.

DISK_CACHE

변수	설명
int nvols_perm	영구타입 볼륨의 개수
int nvols_temp	임시타입 볼륨의 개수
DISK_CACHE_VOLINFO vols[LOG_MAX_DBVOLID + 1]	(DB_VOLPURPOSE purpose, DKNSECTS nsect_free) 볼륨별 목적과 가용 섹터 수, 이 볼륨별 가용 섹터를 기준으로 볼륨별 사전예약을 수행한다.
DISK_PERM_PURPOSE_INFO perm_purpose_info	영구목적 볼륨들 전체의 합산 섹터 정보와 확장 관련 정보를 지니고 있음. DISK_EXTEND_INFO(아래에서 설명)를 유일 변수로 지닌다.
DISK_TEMP_PURPOSE_INFO temp_purpose_info	임시목적 볼륨들 전체의 합산 섹터 정보와 학장 관련 정보를 지니고 있음. 영구목적과는 다르게 nsect_perm_total/free변수를 추가로 지니고 있다.
pthread_mutex_t mutex_extend	볼륨 확장을 위한 뮤텍스
int owner_extend	mutex_extend를 잡은 스레드의 ID

이 중 사전예약 시 예약 가능 여부, 확장의 필요성을 판단하기 위하여 참고하는 변수는 *perm/temp_purpose_info*의 내용이다. 확장 가능한 볼륨들의 확장정보와 함께 전체 볼륨의 가용/전체/최대 섹터 수 등을 저장하고 있다.

DISK_EXTEND_INFO

변수	설명
DKNSECTS nsect_free	볼륨들의 합산 가용 섹터 수
DKNSECTS nsect_total	볼륨들의 합산 전체 섹터 수
DKNSECTS nsect_max	볼륨들의 합산 최대 섹터 수
DKNSECTS nsect_intention	사전예약 시 공간부족으로 추가적인 공간을 확보하려 할 때, 확보하려 하는 섹터 수. 이 값보다는 크게 볼륨을 확장, 추가한다.
pthread_mutex_t mutex_reserve	사전예약을 위한 뮤텍스.
int owner_reserve	mutex_reserve 뮤텍스를 잡은 쓰레드 ID
DKNSECTS nsect_vol_max	볼륨 확장 시 최댓값. 볼륨 생성시 볼륨 헤더의 nsect_max 로 설정된다.
VOLID volid_extend	볼륨 확장 시 auto extend 대상이 되는 볼륨. 하나의 볼륨의 최댓값까지 확장되어야 새로운 볼륨이 생성되므로, 항상 마지막에 생성된 볼륨을 가리킨다.
DB_VOLTYPE voltype	볼륨 타입

이 중 *extend_info*는 개별 볼륨이 아닌 전체 볼륨들의 섹터들에 대한 정보를 합산해서 지니고 있고 temp/perm 두 종류를 나눠서 관리된다. 이 값을 바탕으로 예약할 수 있는지, 확장을 해야 하는지 등을 판단하는 데 임시목적의 데이터와 영구목적의 데이터는 저장되는 볼륨의 종류가 다르므로 이처럼 나누어서 관리한다. 또, 영구목적 볼륨은 영구타입만을 가질 수 있지만, 임시목적 볼륨은 영구타입과 임시타입 모두로 가능하다(디스크매니저 Overview). 이 때문에 임시목적의 purpose_info의 경우는 *DISK_EXTEND_INFO* 외에도 영구목적 볼륨을 위한 정보(*nsect_perm_total/free*)를 지닌다.

> nsect_max는 모든 볼륨의 최대 확장 가능크기의 합으로, 새로운 볼륨이 추가되면 증가한다. 다른 볼륨들은 섹터예약시 공간부족으로 자동으로 확장/추가될 수 있지만 임시목적/영구목적은 이러한 자동확장이 불가능하고 사용자가 임의로 추가(addvoldb)해줘야 한다. 때문에 extend_info없이 nsect_perm_total/free로 예약가능 여부만을 확인한다.

섹터 예약과정

앞서 이야기한 구조체 둘과 함수들을 이용하여 예약이 어떻게 이루어지는지를 살펴보자.

1. *DISK_RESERVE_CONTEXT* 초기화

섹터 예약은 예약요청(*disk_reserve_sectors()*)을 바탕으로 이 context변수를 초기화시켜주고 사전예약(*disk_reserve_from_cache()*)에 들어가는 것으로 시작한다.

```
/* init context */
context.nsect_total = n_sectors;  // n_sectors: 요청 섹터 수
context.n_cache_reserve_remaining = n_sectors;
context.vsidp = reserved_sectors; // 외부에서 메모리할당된 n_sectors만큼의 빈 배열, 예약된 섹터들의 정보를 입력해서 돌려준다. out parameter
context.n_cache_vol_reserve = 0;
context.purpose = purpose;
```

2. 예약목적에따른 *extend_info* 선택

디스크캐시에서 첫 번째로 확인해야 할 것은 무엇일까? 먼저 현재 볼륨들로 요청된 섹터의 예약을 처리할 수 있는지 확인해야 한다. 섹터의 예약을 처리할 수 있다는 것은 현재 볼륨들의 가용 섹터(*extend_info->nsect_free*)가 충분하냐는 것이다. 충분하다면 사전예약을 실행하면 될 것이고, 부족하다면 추가적인 섹터를 확보해야 한다. 임시목적과 영구목적의 데이터는 저장되는 볼륨이 다르므로 각 목적에 맞게 *extend_info*를 선택하여 이를 확인한다.

> 이때, 임시목적의 데이터는 임시볼륨뿐만 아니라 사용자가 addvoldb로 생성한 영구타입/임시목적의 볼륨에도 저장될 수 있다. 때문에 큐브리드는 임시목적의 요청일 경우에 영구타입/임시목적 볼륨에 먼저 섹터 예약을 수행해보고 남은 양에 대하여 임시타입의 볼륨으로 예약요청을 처리한다. 모두 처리된다면 사전예약을 마치고, 아직 모든 요청을 처리하지 못했다면 임시타입의 볼륨으로 다음의 과정으로 넘어간다. 이
>
> > 영구타입/임시목적 볼륨은 사용자가 의도적으로 생성했고, 이후에 과정인 확장과정이 없기 때문에 먼저 확인하고 가능한 만큼 사전예약을 수행한다.

3. 현재 볼륨들의 섹터들로 예약처리 시도

사전예약 요청 수가 현재 볼륨들의 가용 섹터들의 수보다 적어 모든 요청을 처리할 수 있을 것으로 예상되면 현재 볼륨들로 사전예약을 수행한다. 만약 현재 상태로 불가능하게 확실하다면 예약을 시도조차 하지 않고 다음 과정으로 넘어간다.

현재 상태에서의 사전예약처리(*disk_reserve_from_cache_vols()*)란, 요청 타입의 모든 볼륨들을 순회하며 적절한 볼륨에 사전예약 (*disk_reserve_from_cache_volume()*)을 하는 것이다. 이때, *disk_Cache->vols[]* 에 각 볼륨의 목적과 가용 섹터 수가 저장되어 있어, 이를 바탕으로 목적에 맞지 않거나 가용 섹터 수가 너무 적은 경우는 건너뛴다. 볼륨의 가용 섹터 수의 크기가 너무 적을 경우에는 파편화(fragmentation)가 발새할 수 있다. 섹터 추가는 파일로부터 요청되는데, 한 파일 내의 페이지들은 공간 지역성 (space locality)을 가지기 때문에 파편화가 심해질 경우 성능이 저하될 수 있다.

볼륨별 사전예약은 *disk_reserve_from_cache_volume()*를 통해 수행된다. 디스크 캐시의 정보들을 수정(*disk_Cache->vols[volid].nsect_free, extend_info.nsect_free* 감소 등)하고 섹터테이블을 변경하는 작업(Step 2)을 수행하기 위한 정보들(*context->cache_vol_reserve[]* 등)을 생성한다. 마지막으로 예약을 완료한 섹터수 만큼 *disk_reserve_from_cache_volume* 값을 감소시켜 외부에서 추가적인 예약이 필요한지 확인할 수있도록 한다. 이 함수는 사전예약을 수행하는 primitive 함수로 공간 확보 후의 사전예약에서도 사용된다.

4. 현재 상태로는 예약을 처리할수 없다면 공간 확보 후 사전 예약

현재 볼륨들의 상태로는 섹터 요청을 처리할 수 없다면 OS로부터 공간을 추가적으로 할당받는다. 공간 확보방법은 볼륨확장과 새로운 볼륨 추가가 있다. 볼륨 확장 후 섹터를 예약해보고, 여전히 공간이 부족할 경우 추가적인 볼륨을 생성한다.

볼륨확장: 예약요청을 처리할 수 있을 만큼 볼륨의 크기를 확장시킨다. 이 때 요청량이 볼륨헤더에 있는 해당 볼륨의 확장가능한 최댓값보다 크다면 그 값까지만 확장한다. 확장 후에는 확장한 볼륨에 사전예약을 요청한다. 이때, 이전엔 가용 섹터수가 부족하면 사전예약을 시도도 하지 않았던 것과는 다르게 볼륨을 확장한 후에는 가능한 만큼 사전예약을 수행한다. 만약 모든 요청을 처리했다면 사전예약이 종료된다.

볼륨 추가: 남은 모든 섹터의 요청이 처리될 수 있을 때까지 새로운 볼륨을 추가(*disk_add_volume()*)하고 추가한 볼륨에 사전예약(*disk_reserve_from_cache_volume()*)하는 것을 모든 요청을 처리할 수 있을때까지 반복한다.

> 한 볼륨을 최대크기까지 확장하고 나서야 새로운 볼륨을 추가하므로, 확장가능한 볼륨은 항상 하나이다. 이 볼륨을 가리키는 것이 extend_info->volid_extend 변수이다.
>
> 볼륨의 확장과 추가 과정은 별도의 포스트에서 자세히 다루도록 하겠다.

5. 사전예약 정보를 바탕으로 실제 섹터테이블을 변경한다.

3, 4에서 사전예약한 정보를 바탕으로 실제 예약 (섹터테이블 수정)을 수행한다. 사전예약 과정(*disk_reserve_from_cache_volume()*)에서 어떤 볼륨에서 몇개의 섹터를 예약할지를 결정한 상태 (*context->cache_vol_reserve[]*)이고, 이 과정에서 race condition 문제는 모두 해결해 두었으므로 이 정보를 바탕으로 안전하게 실제 예약을 수행한다 (*disk_reserve_sectors_in_volume()*). 볼륨내의 어떤 섹터를 예약할지는 사전예약 시 결정해두지 않았으므로, 각 볼륨의 섹터테이블을 순회하며 사전예약 수만큼 가용 섹터 비트를 찾아 set 해준다. 해당 과정은 STAB Cursor, *disk_stab_iterate_units()*등을 사용하는 것으로 이전 글을 참고하길 바란다.

> volheader->hint_allocset
>
> disk_reserve_sectors_in_volume()을 통해서 섹터테이블을 순회할 때, 처음부터 순회하지 않고 volheader->hint_allocset부터 순회한다. 이는 예약 시 섹터테이블을 차례로 순회하며 처리하고, 끝까지 순회 되지 않은 초기 상태의 섹터테이블이나 파일이 반납한 섹터들이나 모두 연속적으로 있을 확률이 높기 때문이다. 이를 고려하여 예약이 끝난 위치의 뒷부분을 volheader->hint_allocset로 설정하여 다음 예약요청 시 그 위치부터 예약을 시도한다.
>
> ```
> // disk_reserve_sectors_in_volume()
> /* ...(섹터테이블을 순회하며 섹터 예약)... */
> /* update hint */
> volheader->hint_allocsect = (context->vsidp - 1)->sectid + 1;
> volheader->hint_allocsect = DISK_SECTS_ROUND_DOWN (volheader->hint_allocsect); // STAB의 UNIT단위로 내림
> ```

섹터 예약해제

섹터의 예약이 파일의 생성과 추가적인 페이지할당과정에서 발생했던 것 과는 다르게, 예약해제는 파일이 파괴될 때(*file_destroy()*)만 발생한다. 볼륨의 제거나 축소가 없어 예약과정보다 비교적 단순하고, 함수의 길이들도 짧다. 과정을 간단히 요약하자면 다음과 같다.

1. *file_destroy()*에서 파일의 섹터정보를 수집하여 예약해제 요청
2. 수집한 섹터들이 속한 볼륨들을 순회하며 볼륨별 예약해제 요청
3. 각 볼륨마다 커서(cursor)를 사용하여 섹터테이블을 순회하며 유닛(unit)단위로 예약해제

3에서 유닛단위로 이루어지는 예약해제란 예약할 때의 역순으로 섹터테이블의 비트들을 0으로 바꾸고 디스크캐시를 업데이트하여 공간을 확보하는 것을 말한다. 유닛단위로 예약을 해제할 때의 루틴은 영구/임시 목적에 따라 다른데, 임시목적의 경우는 즉시 예약해제를 하지만 영구목적의 경우는 *log_append_postpone()*을 통해 트랜잭션이 commit(*log_commit()*)될 때까지 해당 작업을 미룬다. 이는 데이터베이스 예약 해제되는 섹터들이 트랜잭션이 커 밋되 기전까지는 다른 트랜잭션에 의해 사용되는 일이 없도록 하기 위함으로 이후에 트랜잭션 혹은 로그를 다루는 포스트에서 자세히 다루도록 한다.

> 관련 함수:
>
> file_destroy()
>
> disk_unreserve_ordered_sectors()
>
> disk_unreserve_ordered_sectors_without_csect()
>
> disk_unreserve_sectors_from_volume()
>
> disk_stab_unit_unreserve()

그 밖에

임시목적 볼륨과 로깅 (Logging)

코드를 분석하다 보면 임시/영구 목적에 따라 분기되는 코드들을 자주 만날 수 있다. 관련 정보가 따로 관리되는 부분에서도 분기되지만, 디스크에 쓰는 작업이 발생할 때 항상 분기되어 로딩(logging) 여부를 결정한다. 임시목적의 경우 로깅도 하지 않고 commit 될 때까지 연산을 미루는 일도 없다. 살펴본 내용 중에는 예약이나 예약해제 과정의 마지막 부분(*disk_stab_unit_[un]reserve()*)에서 섹터테이블을 변경하는 때에 분기되는 것을 확인할 수 있다.

Disk Cache Sync

두 단계로 나누어진 예약과정은 atomic 하지 않고 이 둘을 동기화시켜주는 락(Lock) 또는 래치(Latch)도 존재하지 않는다. 즉, 사전예약을 끝내고 섹터테이블을 순회하며 변경하는 동안에 캐시와 실제 디스크 내용이 다른 상태를 지닐 수 있다. 하지만 예약할 때는 항상 캐시->섹터테이블 순으로 수정하고, 예약해제 시에는 항상 섹터테이블->캐시 순으로 수정하기 때문에 섹터 예약의 기준이 되는 디스크 캐시는 항상 섹터테이블보다 적은 양의 가용 섹터를 지닌다. 따라서 이러한 불일치는 섹터 예약 시 문제를 발생시키지 않는다.

Disk Check

예약과정에서는 이러한 불일치가 문제가 되지 않지만, 디스크 캐시와 볼륨 내의 정보는 언젠가 일치되어야 할 정보이다. 이를 확인하기 위해 *disk_check()*란 함수가 존재하는데, 이때는 두 단계의 예약과정을 atomic 하게 보고 consistency를 확인해야 할 것이다. 예약과정에서 확인할 수 있는 *csect_enter_as_reader()*등이 이를 위한 것이다.