큐브리드 파일 페이지 할당
(CUBRID File Page Allocation)

Posted by : Jaeeun Kim at Aug 5, 2019

페이지 할당은 어떻게 이루어질까?

- CUBRID File Page Allocation -

이번 글에서는 이전에 살펴본 파일의 구조를 바탕으로 파일매니저의 주된 역할인 페이지할당이 어떻게 이루어지는지 살펴본다. 페이지 할당과정은 디스크매니저의 섹터예약과정과 유사하며, 좀 더 단순하다. 페이지할당 과정을 이해하기위해서는 파일내의 페이지할당여부를 관리하는 파일테이블에 대하여 알아야 한다. 이를 바탕으로 페이지 할당이 어떤식으로 이루어지는지 살펴보자. 또한 임시파일의 경우에는 어떻게 이러한 과정이 단순화될 수 있는지도 임시파일의 페이지할당과정과 함께 살펴본다.

이 포스트는 다음과 같은 내용을 다룬다.

- 파일 테이블
- 페이지 할당
    - 영구 파일
    - 임시 파일
- 페이지 할당 해제

파일 테이블 페이지

파일테이블은 파일내의 페이지들을 관리하기 위한 테이블로 세가지 종류가 있다. 예약한 섹터들의 할당정보를 Partial Sectors Table로 관리하며, 테이블의 엔트리는 각 세터별 예약여부에 대한 비트맵이다. 모든 섹터가 예약되면 비트맵은 더 이상 소용없으므로 비트맵없이 Full Sectors Table로 옮겨진다. 이 둘이 나뉘어져 있는 이유는 할당가능한 페이지를 빠르게 탐색하기 위한 것으로 보인다. User Page Table은 Numerable 속성을 위한 페이지들의 할당 순서를 저장한다.

파일테이블의 종류

- **Partial Sectors Table**: 파일이 예약한 섹터중에서 섹터 내의 모든 페이지가 할당되지 않은 섹터들의 정보를 담고 있다. 테이블의 엔트리는 *FILE_PARTIAL_SECTOR*로 각 섹터의 ID(*VSID*)와 할당여부에 대한 비트맵(*FILE_ALLOC_BITMAP*, UINT64)을 가지고 있다.
- **Full Sectors Table**: 파일의 예약한 섹터 중 섹터 내의 모든 페이지가 할당된 섹터들의 정보를 담고 있다. 테이블의 엔트리는 각 섹터의 ID(*VSID*)이다.
- **User Page Table**: 위의 둘과는 다르게 페이지를 할당 후 할당된 페이지들을 할당된 순서대로 담고 있다. 테이블의 엔트리는 각 페이지의 ID(*VPID*)이다. Numerable File의 경우에만 이러한 테이블이 생성된다.

파일의 종류에 따른 파일헤더의 파일테이블구성

앞서글들에서 언급한 것처럼 파일 헤더내의 각 파일테이블들의 크기는 파일의 타입과 Numerable속성유무에 따라 상이하다. 이는 아래와 같다. 그림 옆의 숫자는 파일헤더를 제외한 파일헤더페이지내에서 각 파일테이블이 차지하는 공간의 비율을 나타낸다. 이 때 파일헤더는 고정크기이다.

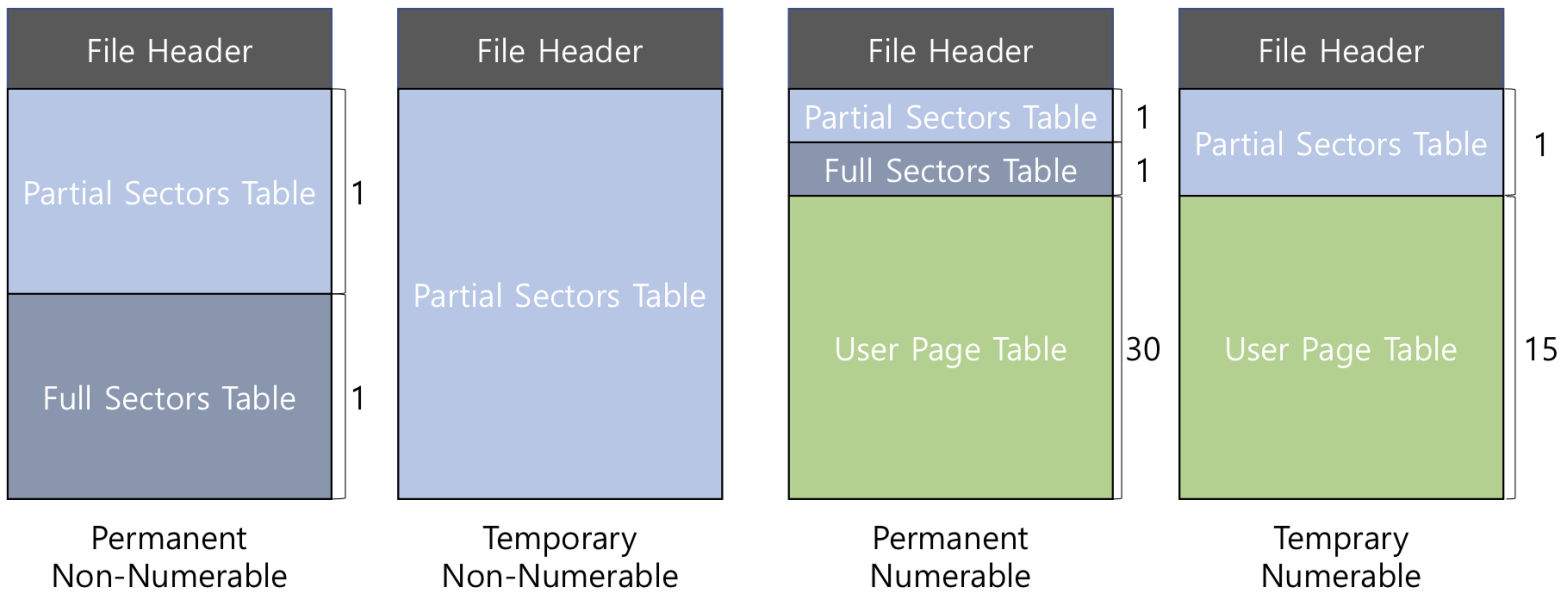

Figure 1: 파일종류에 따른 파일헤더의 파일테이블들

임시파일의 경우 페이지 할당해제 자체가 없고, 순차적으로 할당을 진행하므로 Full Sectors Table이 필요 없고, 영구파일의 경우 이를 2등분하여 Partial/Full 섹터 테이블을 만든다. 이 때, 파일 헤더내에서 각 파일테이블이 차지하는 크기만 다를 뿐 추가적으로 생성되어 연결되는 파일테이블들은 여전히 한 페이지씩을 차지한다.

동적인 크기의 파일테이블페이지

파일테이블 페이지의 경우 볼륨의 섹터테이블과는 다르게 크기가 유동적이다. 볼륨의 경우는 최대크기가 정해져 있고 최대크기를 커버할 수 있는 만큼만 섹터테이블이 존재하면 되므로 볼륨이 최초에 생성될 때 고정된 양의 페이지를 사용하여 섹터테이블을 생성하지만, 파일의 경우는 파일의 확장과 사용도에 따라 파일테이블을 추가로 할당 받아야 한다. 또, 필요한 순간에 추가로 파일테이블페이지를 할당받으므로 이들은 물리적으로 연속적이지 않고 흩어져 있으며, 포인터로 연결되어 있다.

앞서 말한 파일테이블에 관한 내용을 그림으로 정리하면 다음과 같다. 한 파일내에서 유저페이지를 제외한 시스템 테이블 페이지(파일헤더페이지를 포함한)들의 관계를 개념적으로 표현한 것이다. 물리적으로 파일테이블페이지들은 유저페이지와 함께 볼륨내 임의의 위치에 할당되어 있다. 파일헤더페이지는 파일헤더와 세가지 종류의 테이블이 들어 갈 수 있으며 파일의 사용도에 따라 각 테이블은 추가적인 파일테이블페이지와 연결된다.

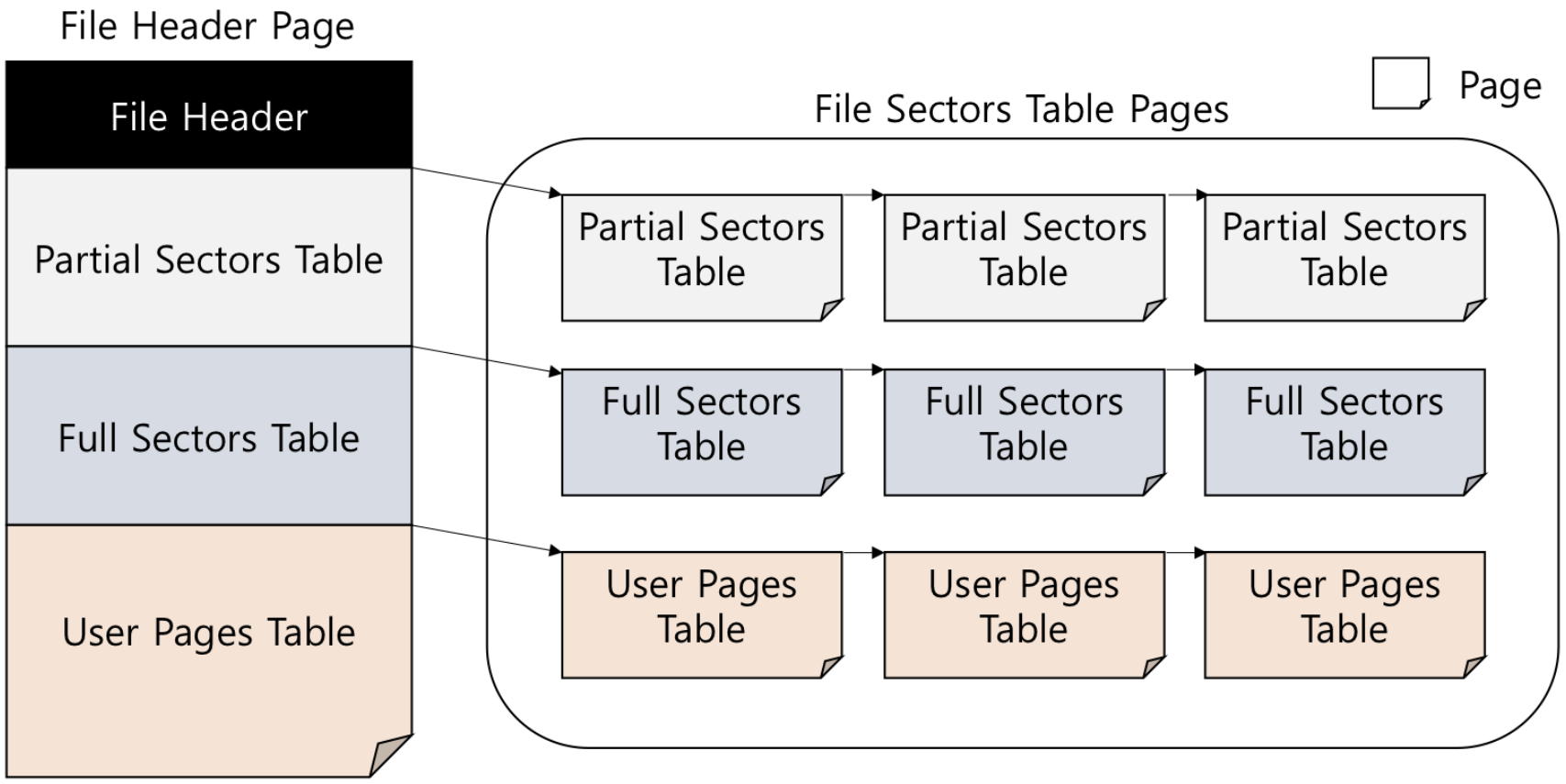

Figure 1: 파일테이블 구조

참고로, 파일테이블들을 모두 순회해보면 어떤 섹터의 어떤페이지가 현재 파일에 할당되어 있는지를 알 수는 있지만, 파일테이블 순회없이 파일내 임의의 페이지에 바로 접근할 수 있는 방법은 없다. 만약 파일의 용도에 따라 할당받은 페이지들을 이후에 빠르게 접근하고 싶다면 그러한 기능은 해당 파일타입이 직접 제공해야한다. 예를들어 힙파일(Heap FIle)의 경우에는 다음 힙페이지로의 링크를 레코드로 저장해둔다. 파일매니저나 파일테이블의 역할은 단순히 페이지의 할당까지만이다.

File Extendible Data

파일테이블은 File Extendible Data라는 구조로 이루어져 있다. 이 구조는 단순히 배열들의 리스트와 같은 형태로, 이론적으로 무제한의 데이터를 여러 디스크 페이지들 안에 담고 이를 연결시켜주는 일반화된 포맷이다. 이 구조는 파일테이블들(Partial/Full Sectors Table, User Page Table), 파일 트래커(File Tracker)등에서 사용된다.

File Extendible Data Format

구조는 단순하다. 헤더와 아이템(Item, 데이터)의 셋(set)으로 이루어진 컴포넌트들의 리스트(Singly Linked List)이다. 이를 도식화해보면 다음과 같다.

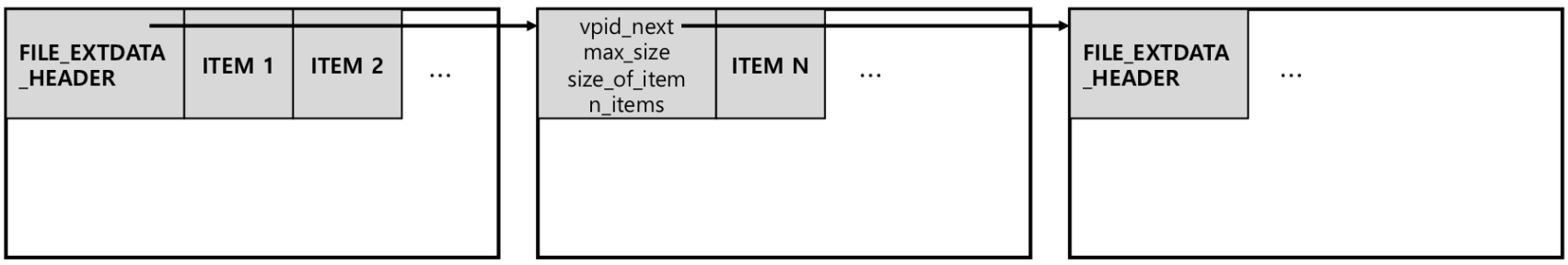

Figure 1: File Extendible Data Format

헤더(*FILE_EXTDATA_HEADER*)는 헤더가 포함된 컴포넌트의 아이템의 갯수(*n_items*), 아이템의 크기(*size_of_item*), 컴포넌트의 최대 크기(*max_size*), 다음 컴포넌트로의 링크(*vpid_next*)를 가지고 있다. 여기서 각 변수들은 모두 헤더가 포함된 컴포넌트내의 정보이다. 즉, 전체 아이템의 갯수를 알기 위해서는 모든 컴포넌트의 헤더를 순회하며 *n_item*을 모두 더해야 한다. 이전 글의 파일테이블 구조에서 살펴본 여러 페이지에 흩어져 있는 파일 테이블들이 연결되어 있게 해주는 구조가 이 File Extendible Data이다. 페이지 ID(*vpid*)를 통해 연결되므로 첫 컴포넌트는 디스크 페이지내의 어떤 위치에 들어가도 상관 없지만, 이 후의 컴포넌트들은 각 디스크 페이지의 첫 부분에 컴포넌트가 존재해야한다.

File Extendible Data Operations

관련 연산은 Item append, insert, remove, update, merge, search, size 등 일반적인 Container류 자료구조의 연산들로 이루어져 있으며 다음과 같은 특성을 지닌다.

- remove, insert 등은 항상 삽입/제거 위치 뒤에 있는 모든 데이터의 memmove()연산이 일어남
- merge, search 등은 Item들이 ordered, unordered이냐에 따라 분기된다. ordered일 경우 search시 이진탐색이 가능하다.

각 연산들의 구현은 대부분 포맷과 같이 단순하여 설명을 생략하고 필요할 경우에는 사용되는 곳에서 함께 이야기하도록 한다.

> 탐색을 위해 VSID로 정렬되어 있는 Full/Partial Sectors table이 ordered의 예이다.

File Table Item

File Extendible Data의 각 아이템은 어떠한 포맷이라도 올 수 있다. 파일 테이블 세종류 모두 이러한 구조를 통해 이루어져 있지만 아이템은 각각 다르다. 단순히 예약여부, 할당여부를 판단하는 경우 (Full, User)는 ID만을 가지고 있는 반면에 각 섹터들내의 할당여부를 트래킹하는 경우(Partial)는 섹터내의 페이지들에 대한 비트맵을 가지고 있다.

- Partial Sectors Table: *FILE_PARTIAL_SECTOR*(*VSID*, *FILE_ALLOC_BITMAP*)
- Full Sectors Table: *VSID*
- User Page Table: *VPID*

페이지 할당

파일매니저는 디스크매니저로부터 예약한 섹터들을 파일이 필요로 할 경우 페이지단위로 공간을 할당한다. 영구파일과 임시파일은 페이지 할당을 위하여 페이지를 관리하는 방법도 다르고, 페이지를 할당하는 방법도 다르다. 다음의 공통 인터페이스(*file_alloc()*)를 통해 파일타입에 따라 각각 *file_alloc_perm(), file_alloc_temp()*로 분기되어 페이지를 할당 한다.

```
int
file_alloc (THREAD_ENTRY * thread_p, const VFID * vfid,
            FILE_INIT_PAGE_FUNC f_init, void *f_init_args,
            VPID * vpid_out, PAGE_PTR * page_out)
```

페이지가 할당된 후에는 Numerable 파일의 경우에는 User Page Table에 페이지를 추가한 후, 인자로 받은 페이지 초기화 함수(*f_init*)을 통해 디스크페이지의 내용을 초기화한다. 이 때 이 초기화 함수는 각 파일의 타입(HEAP, BTREE 등)에 맞게 페이지를 포매팅한다.

영구파일의 페이지 할당

큐브리드 파일의 페이지 할당은 디스크 매니저의 섹터예약과 유사하다. 할당과정 자체는 유사하면서 파일은 볼륨과 다르게 논리적인 단위이기에 좀 더 단순하다. 페이지 할당은 기본적으로 한 페이지씩 수행하며, 파일테이블에서 할당되지 않은 페이지를 찾아 할당 표시를 한다. 영구파일의 페이지 할당의 주요 과정은 다음과 같다. (*file_perm_alloc()*)

1. 파일헤더를 확인하여 파일이 예약한 섹터 중 가용 페이지(fhead->n_page_free)가 없다면 파일(*file_perm_expand()*)을 확장한다.
2. 파일헤더의 Partial Sectors Table에 아이템이 하나도 없다면, 링크로 연결된 다음 파일테이블에서 파일헤더의 파일테이블 크기만큼을 가져온다 (*file_table_move_partial_sectors_to_header()*).
3. 파일헤더의 Partial Sectors Table의 첫 번째 아이템(*file_extdata_start()*)에 페이지 할당 비트를 하나 SET 한다 (*file_partsec_alloc()*).
4. 파일헤더의 Partial Sectors Table에서 페이지를 할당한 섹터아이템의 모든 페이지가 할당되었으면 해당 아이템을 Full Sectors Table로 옮긴다.
5. 유저페이지를 할당하였고 Numerable파일이라면 User Page Table에 새로운 아이템을 추가(*file_numerable_add_page()*)한다.

1에서 *file_perm_expand()*는 디스크매니저에게 추가적인 섹터예약을 요청(*disk_reserve_sectors()*)한다. 이 때, 확장크기 정보는 파일 헤더의 *tablespace*에 담겨 있고, 이때 파일헤더의 Partial Sectors Table을 넘어가지 않을 정도로만 확장한다. 이 크기를 넘어갈 경우 추가적인 파일테이블 페이지 할당 등 로직이 복잡해져 제한을 둔 듯하다.

2,3에서 항상 Partial Sectors Table의 첫번째 아이템에서 페이지를 할당하는 것을 보면 알 수 있듯이, 페이지 할당시 이 테이블은 마치 Queue자료구조처럼 가장 앞의 아이템만이 사용된다. 해당 아이템의 모든 페이지가 할당되었을 경우 4에서 바로 Full Sectors Table로 옮겨지므로 항상 첫번째 아이템에는 할당되지 않은 페이지가 있음을 보장할 수 있다.

또한 2에서 다음 파일테이블의 모든 아이템을 가져올 경우 해당 파일테이블 페이지는 할당해제되므로 파일내에 할당되지 않은 페이지가 존재할 경우, 항상 파일헤더페이지에 연결된 다음 Partial Sectors Table 페이지에는 할당가능한 섹터가 존재함이 보장된다. 또한, 이렇게 파일테이블 페이지가 할당해제되는 경우에는 해당 페이지를 페이지할당에 재사용한다 (3, 4과정을 건넘 뛈).

임시파일의 페이지 할당

임시파일의 페이지 할당(*file_temp_alloc()*)은 영구파일의 페이지 할당보다 훨씬 단순한다. 이는 일시적인 데이터를 위한 파일이라는 특성 덕분이다. 영구파일에서 Partial/Full로 섹터를 분류해서 관리하는 것은 이들을 함께 관리하면 시간이 지남에 따라 페이지의 할당과 해제가 반복되면서 점점 가용페이지를 찾는것의 비용이 커질 것이기 때문이다. 하지만 임시파일의 경우 일시적으로 사용하는 파일이므로 단순히 그 순간에 필요한 만큼 페이지를 할당받아 사용하고, 사용이 끝나면 파일 자체를 제거한다. 따라서 다음과 같이 페이지를 관리한다.

- Partial/Full Sector Table을 구분해서 사용하지 않는다.
- Partial Sector Table형태의 파일테이블 하나만을 사용하며 순차적으로 섹터내의 페이지를 할당해간다.
- 순차적으로 페이지를 할당하므로 예약을 위하여 마지막에 예약한 위치를 파일헤더에 캐시해두고 다음 할당시 이 위치에서 바로 할당을 수행한다. 파일헤더의 *vpid_last_temp_alloc, offset_to_last_temp_alloc* 변수들이 이를 위한 캐싱변수들이다.
- 페이지 할당 해제를 하지 않는다.

임시파일의 예약과정은 이러한 특성을 그대로 반영한 코드로 단순하여 자세한 설명은 생략한다.

페이지 할당 해제

임시파일의 경우 페이지할당해제과정이 없으므로 영구파일의 경우에 대해서만 살펴본다.

페이지의 할당해제(*file_dealloc()*)도 섹터의 예약해제와 같이 *log_append_postpone()*를 이용하여 즉시수행하지 않고, 트랜잭션이 완료될때까지 작업을 미룬다. 트랜잭션이 완료될때 수행되는 작업(*file_rv_dealloc_internal(), file_perm_dealloc()*)은 할당할 때와 반대로 해당 페이지가 속한 파일 테이블 (Full/Partial)을 찾아서 속하는 섹터의 비트는 OFF하는 것이다. 코드를 봤을때 복잡한 이유는 다음과 같이 예외상황들이 많기 때문이다.

- Partial Sectors Table에서 해당 페이지가 속하는 섹터 아이템을 찾았을 경우는 bit OFF 후 끝
- Full Sectors Table에서 찾았을 경우 할당해제하는 페이지를 제외하고 모든 비트가 set된 비트맵을 추가하여 Partial Table로 이동시켜야 함.
- Full Sectors Table에서 제거하는 과정에서 해당 아이템이 마지막 아이템이라면 Full Sectors Table이 있는 페이지를 함께 할당해제 해줘야 함.
- Partial Sectors Table에 추가하는 과정에서 만약 빈공간이 없으면 새로운 Partial Sectors Table Page를 할당해줘야 함.

파일 테이블을 조정한 후에는 *pgbuf_dealloc_page()*를 통해 버퍼페이지를 초기화시키고 LRU 3 ZONE으로 이동시켜 후에 Flusher에 의해 할당해제되었다는 정보가 디스크에 써질 수 있도록 한다.

버퍼 페이지 (Buffer Page)

버퍼페이지는 디스크페이지를 메모리에 적재시키기위한 버퍼 공간이다. 디스크페이지들은 항상 이 버퍼페이지를 통해 접근(fix)되며, 변경된 페이지 내용은 버퍼매니저의 정책에 따라 디스크와 동기화 된다. 디스크페이지의 할당해제시 페이지의 타입을 UNKNOWN으로 변경하여 해제된것을 표시하고, LRU3 ZONE으로 이동시켜 버퍼매니저에 의해 최대한 빨리 재사용될 수 있도록 한다.

이는 차후에 버퍼매니저에 대한 글을 쓰게될 경우 자세히 다루도록 하겠다.

유저페이지를 할당해제 할 경우 Numerable이라면 User Page Table에서 제거한다. 이 때 바로 제거하지 않고 delete 표시만을 해두고 차후에 제거한다.