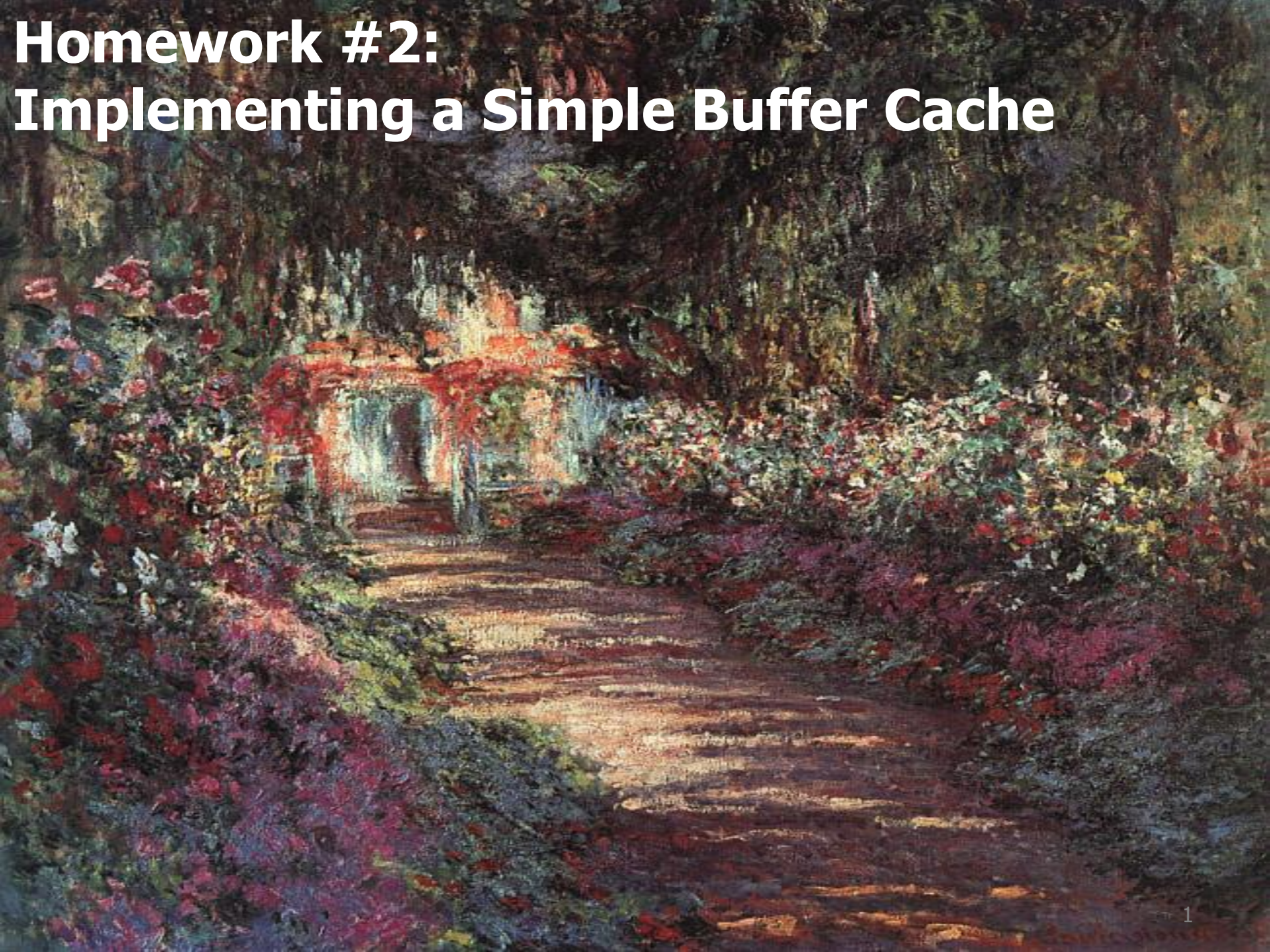
Homework #2: Implementing a Simple Buffer Cache



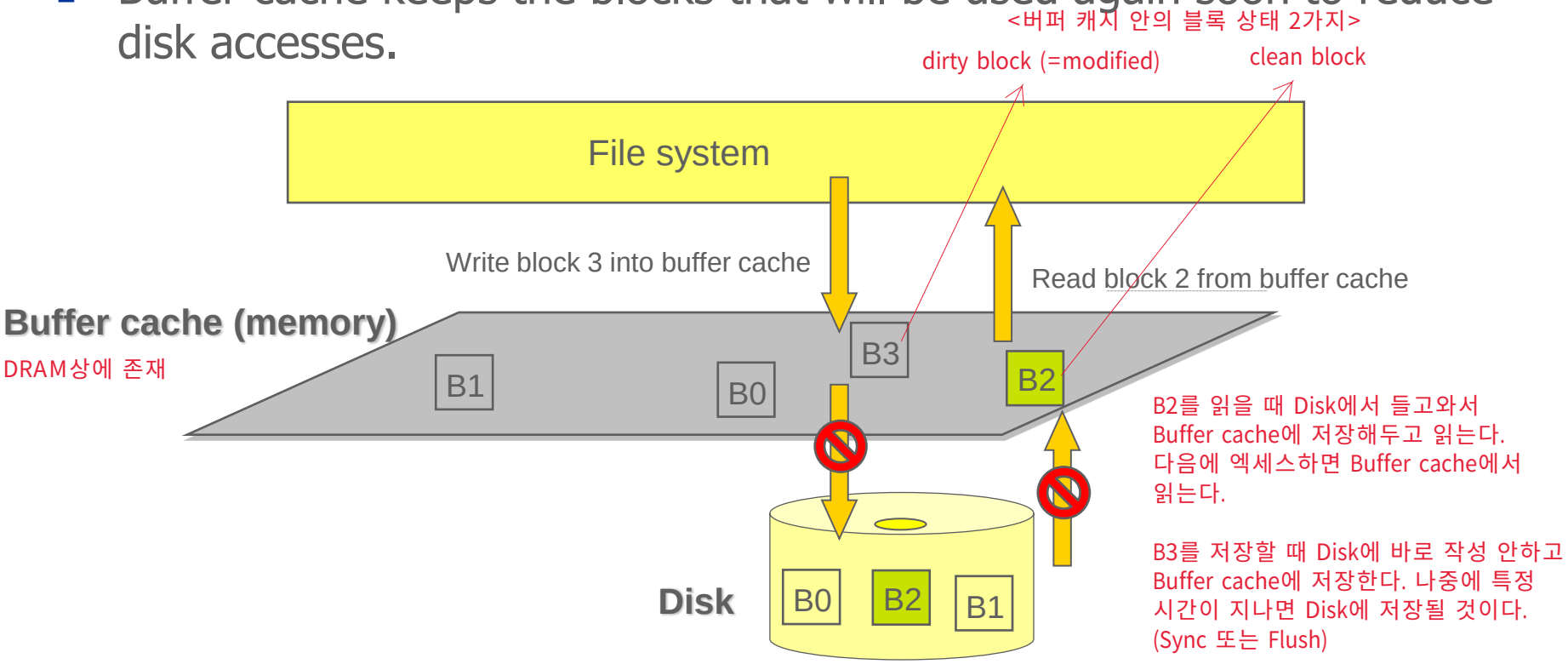
Buffer Cache

버퍼캐시를 구현해보자!

한번 접근한 데이터 or 블록 들은 가까운 미래에 다시 접근될 가능성이 높아
Disk I/O 를 최소화 하기 위해 한번 접근한 것은 버퍼캐시에 저장을 해두고 다음에 접근할 때
Disk가 아닌 Buffer Cache에 접근하는 것이다.

- Motivation
  - Whenever files are accessed, their data should be fetched from disks. However, the disk accesses incurs large I/O overhead. On file access patterns, data that are accessed once will be used again soon. That is, file accesses have strong localities.
- Buffer cache keeps the blocks that will be used again soon to reduce disk accesses.

<버퍼 캐시 안의 블록 상태 2가지>

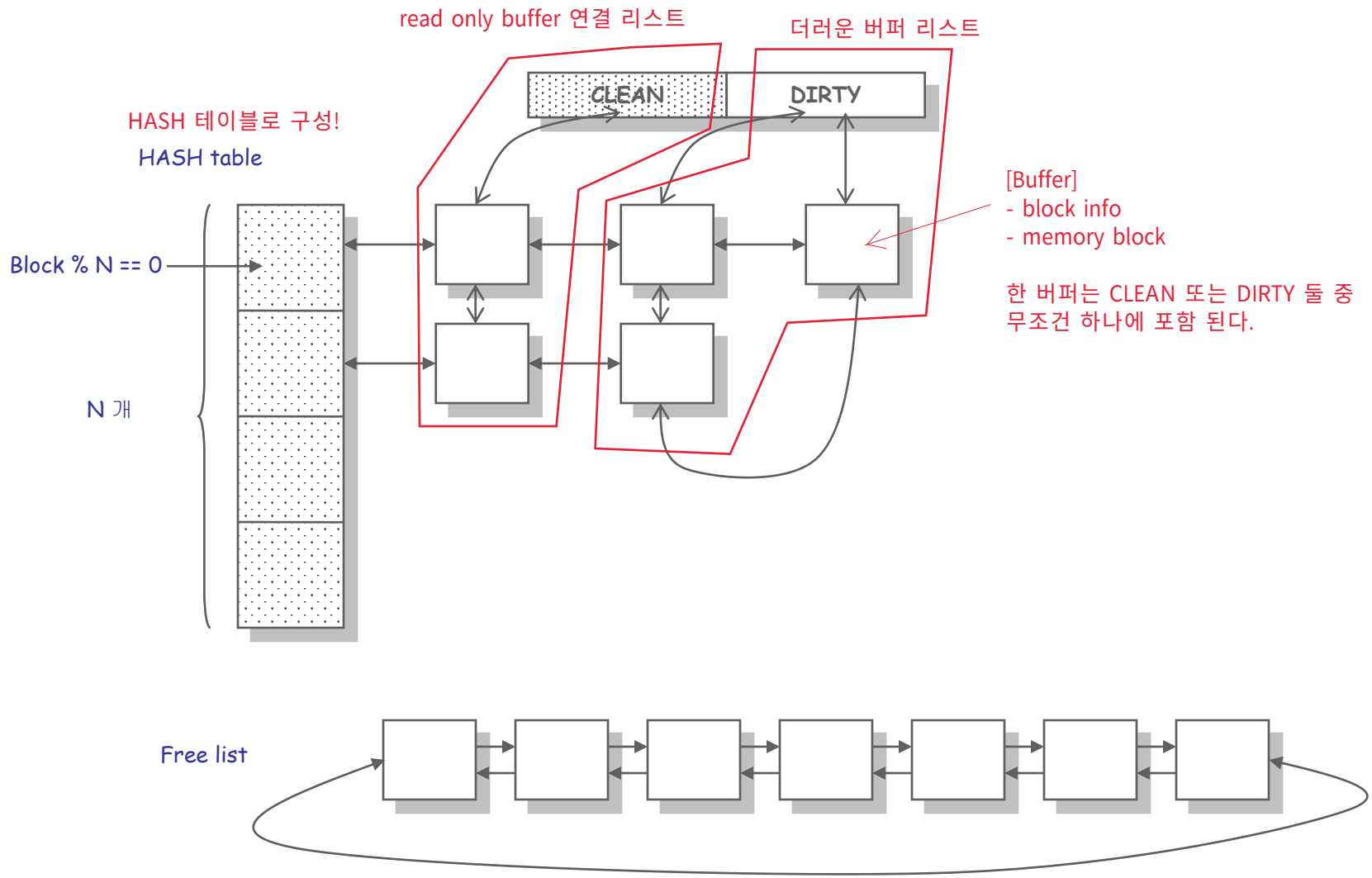


B2를 읽을 때 Disk에서 들고와서 Buffer cache에 저장해두고 읽는다. 다음에 엑세스하면 Buffer cache에서 읽는다.

B3를 저장할 때 Disk에 바로 작성 안하고 Buffer cache에 저장한다. 나중에 특정 시간이 지나면 Disk에 저장될 것이다. (Sync 또는 Flush)

Buffer Cache in Commercial OS

상용 OS의 버퍼 캐시

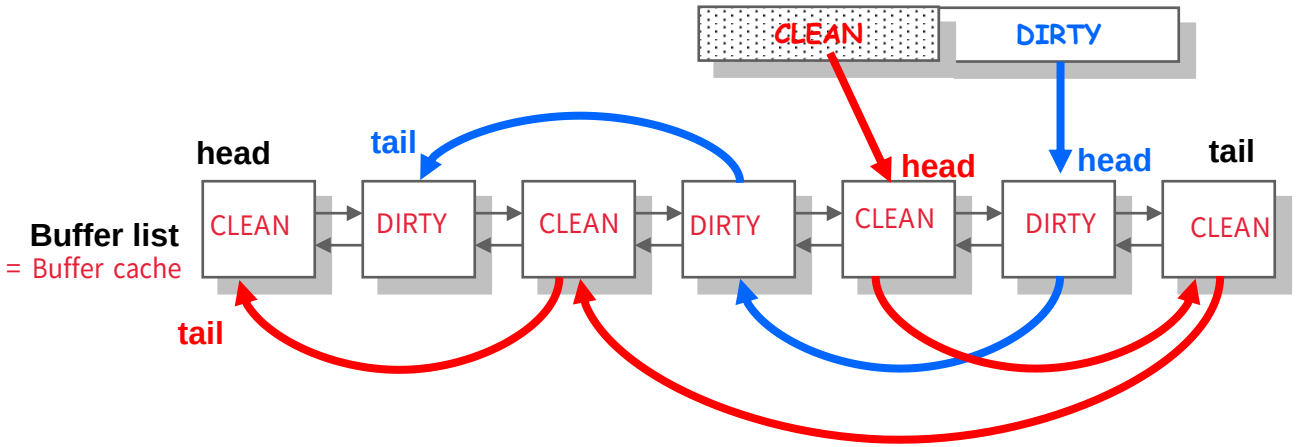


Buffer Cache Structure (Cont'd)

- Clean list
  - A list of buffers that stores read-only blocks
- Dirty list
  - A list of buffers to store blocks that are modified by write system call.
- Hash table or queue 이번 과제에서 버퍼 갯수는 열 몇개.
그래서 해시 테이블을 사용할 필요는 없다.
  - Implemented to quickly look up a buffer that stores a given block.
  - An array of the number of N entries.
  - The *i* th entry points the list of buffers that store blocks meeting the following condition:
    - Physical block no % N == i
- However, you should take much time to implement the three lists. Therefore, a double linked list is replaced with the hash queue.

Buffer Cache Simplified for YOU!!!

우리는 Double Linked List 로 구현하면 된다!



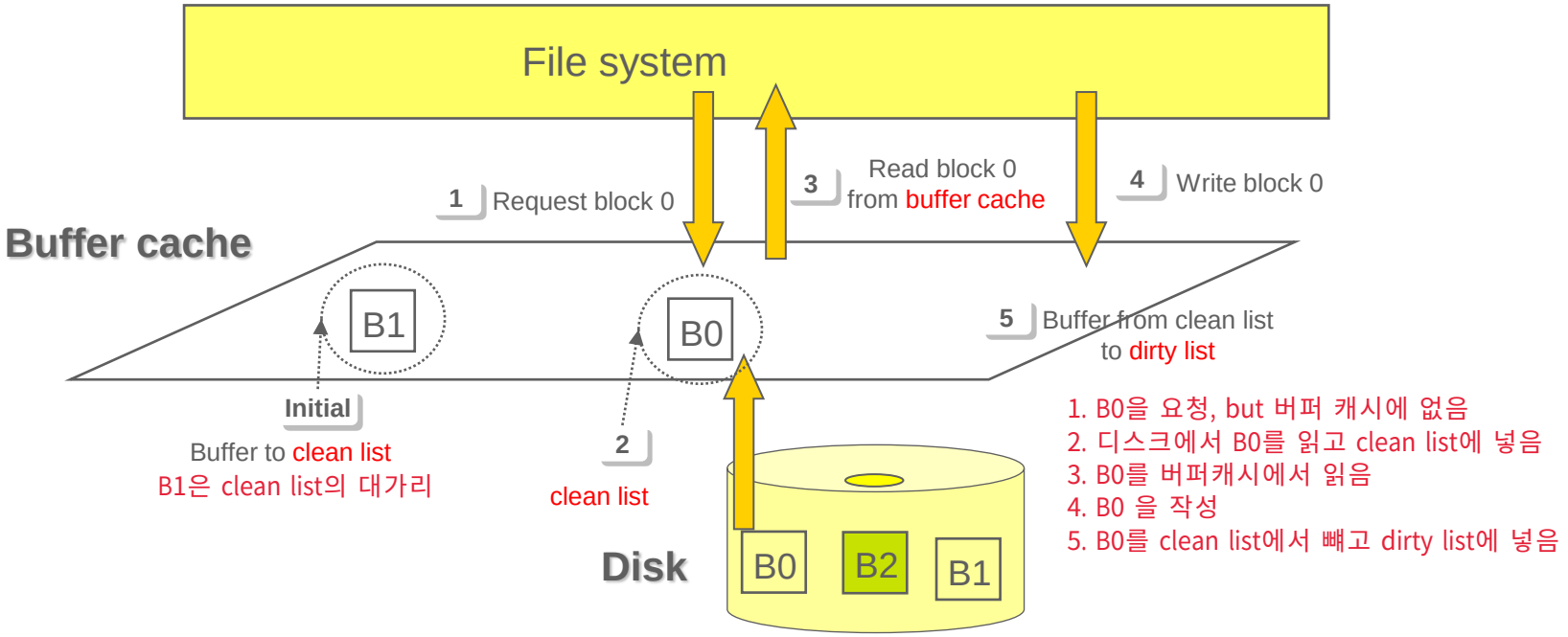
Buffer list : 0 - 1 - 2 - 3 - 4 - 5 - 6

Clean list : 4 - 6 - 2 - 0

Dirty list : 5 - 3 - 1

Example: Buffer Cache Operations

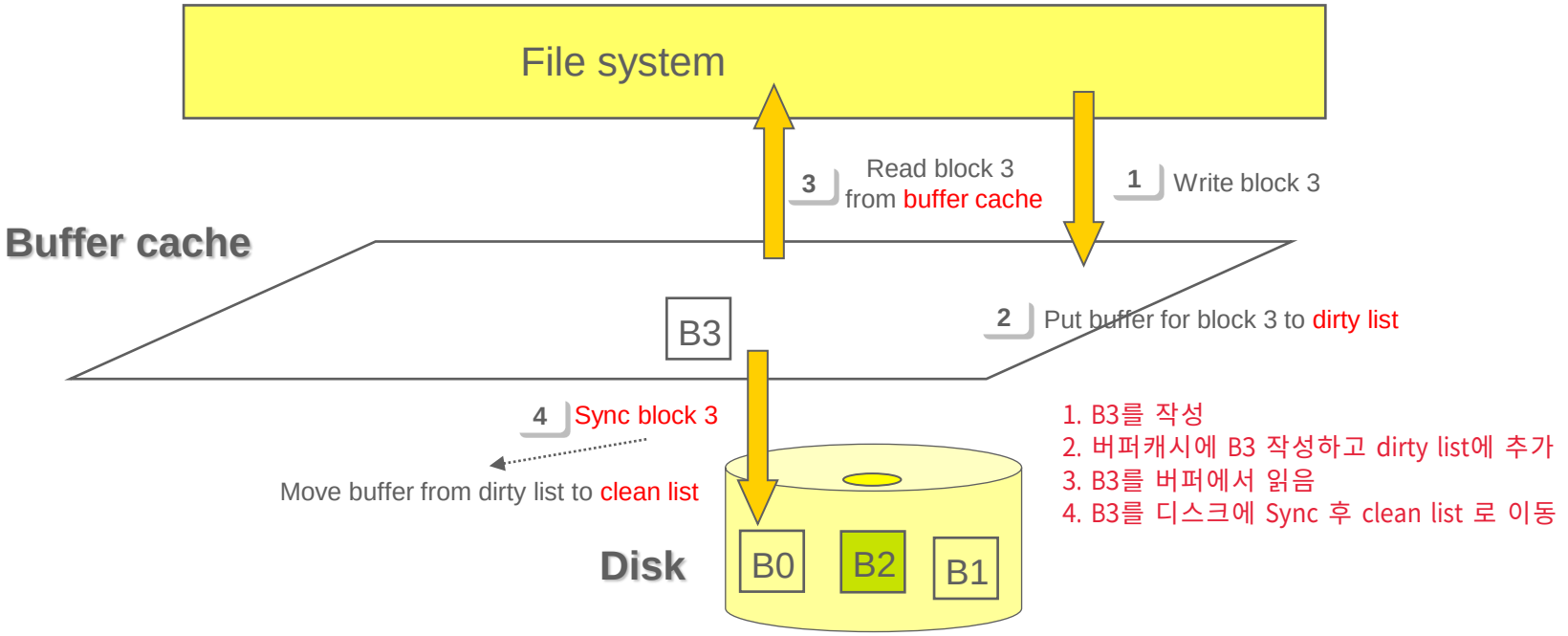
- Read a block from disk → Re-read the block in buffer cache → write the block in buffer cache



- Two functions is given for read/writing a block from/to disk.
  - DevReadBlock(int blkno, char* pData);
  - DevWriteBlock(int blkno, char* pData);

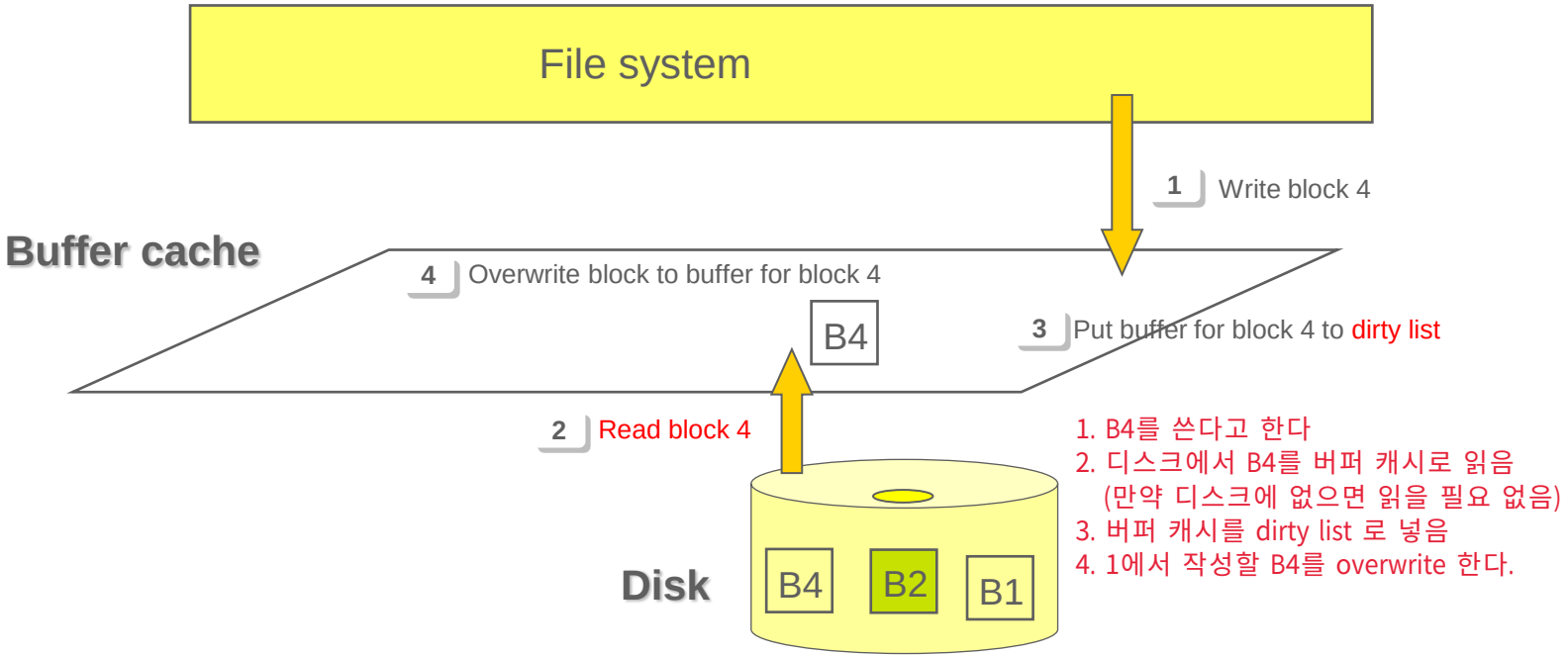
Example: Buffer Cache Operations

- Write a new block → Sync(or flush) the block to disk
  : there is not block 3 in disk



Example: Buffer Cache Operations

- Write a block
  : block 4 is stored in disk, but is not in buffer cache



Buffer Structure

버퍼 구조체

Buffer structure

```
#define MAX_BUFLIST_NUM (2)
#define MAX_BUF_NUM     (8)
#define BLKNO_INVALID   (-1)

typedef struct Buf Buf;
typedef enum __BufState
{
    BUF_STATE_CLEAN,
    BUF_STATE_DIRTY
} BufState;
typedef enum __BufList
{
    BUF_LIST_DIRTY = 0,
    BUF_LIST_CLEAN = 1
} BufList;
struct Buf
{
    int                 blkno;
    BufState            state;
    void*               pMem;
    TAILQ_ENTRY(Buf)    blist;
    TAILQ_ENTRY(Buf)    slist;
    TAILQ_ENTRY(Buf)    llist;
};
```

더티인지 클린인지 2가지

전체 버퍼 갯수 (testcase 에서 바뀔 수도 있음)

블록 번호

CLEAN or DIRTY

1개의 블록을 저장할 수 있는 메모리

Buffer list

Dirty or Clean list

LRU list

Double linked list 를 쉽게 구현하기 위해 나온 매크로가 있음.

Buffer structure

```
TAILQ_HEAD(bufList, Buf)
    pBufList = NULL;
TAILQ_HEAD(stateList, Buf)
    ppStateListHead[MAX_BUFLIST_NUM] = {NULL,};
TAILQ_HEAD(lrulList, Buf)
    pLruListHead = NULL;
```

버퍼 리스트

더티인지 클린인지 배열

Lru 는 나중에 나옴

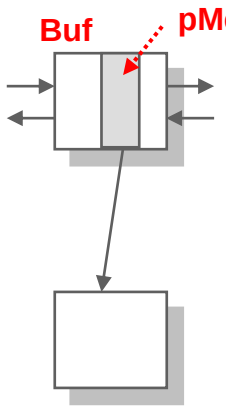
TAILQ man page

https://man7.org/linux/man-pages/man3/tailq.3.html

TAILQ example:
https://tttsss77.tistory.com/18 (한글 버전)
https://blog.jasonish.org/2006/08/19/tailq-example/
https://seokbeomkim.github.io/posts/bsd_queue/

Buffer 구조

- Buffer (or Buf) = Block Control block
  버퍼는 블록 컨트롤 블럭이다.
  - File block을 저장하는 메모리 공간 + block의 정보를 저장하는 변수



Block을 저장하는 메모리 공간 (블록 크기의 메모리 공간)
> disk에서 read된 블록을 여기에 저장한다.
> disk로 write하고자 하는 블록을 여기에 저장한다.
> 메모리 공간은 malloc(BLOCK_SIZE)로 할당함

- 전체 Buffer 개수
  - 특정 개수 N을 정해진다.
  - N개 이상의 블록을 저장하고 싶다면 block replacement를 수행한다

Buffer cache에서 block 읽기

- Void BufRead(int blkno, char* pData) 내가 구현해야하는 함수
  - Blkno: 읽을 block 번호
  - pData: 블록을 담을 메모리 공간
  - 디스크에서 블록을 읽는 기능을 DevWriteBlock 대신 BufRead가 담당한다.

Disk I/O without Buffer Cache

```
int blkno = 1;
char pData[BLK_SIZE];

DevReadBlock(blkno, pData);
pData[10] = 10; // modify
DevWriteBlock(blkno, pData);
```

Disk I/O with Buffer Cache

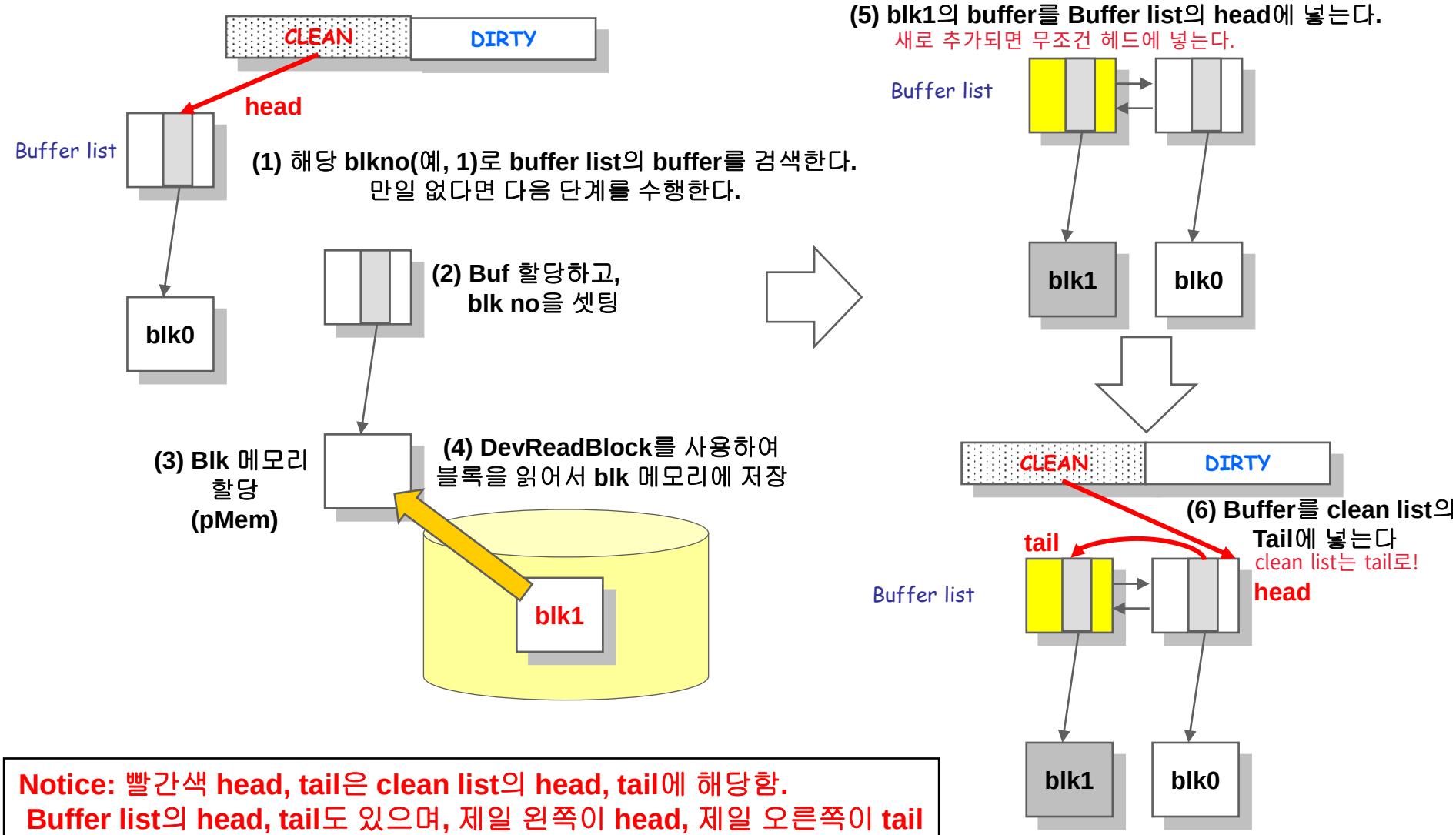
```
int blkno = 1;
char pData[BLK_SIZE];

BufRead(blkno, pData);
pData[10] = 10; // modify
BufWrite(blkno, pData);
BufSync();
```

좌, 우 둘 코드를 거의 비슷하다.

Buffer cache에서 block 읽기

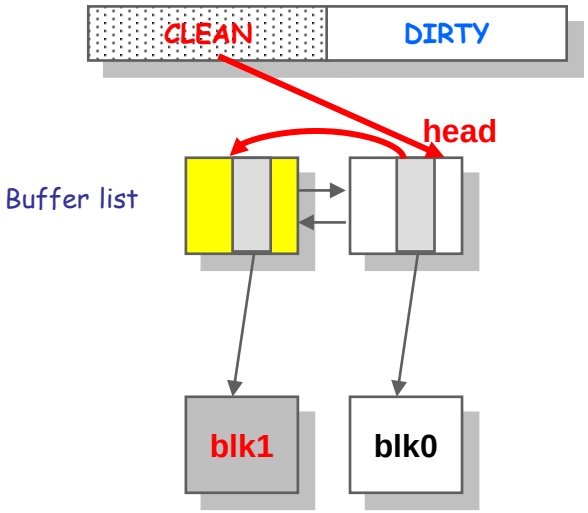
BufRead() 함수 구현



Notice: 빨간색 head, tail은 clean list의 head, tail에 해당함.
Buffer list의 head, tail도 있으며, 제일 왼쪽이 head, 제일 오른쪽이 tail

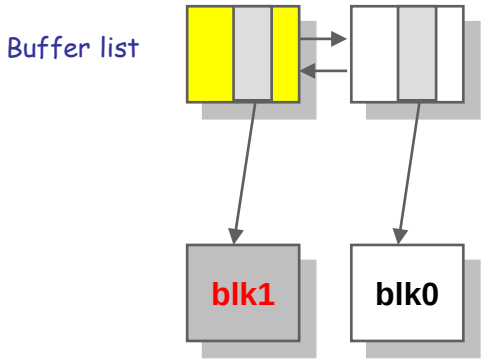
Buffer cache에서 block 읽기

- 해당 block이 buffer list에 있다면?
  - 디스크에서 읽지 않고 buffer list의 해당 buffer에서 읽음 → disk I/O 감소
- BufRead(1, pData)

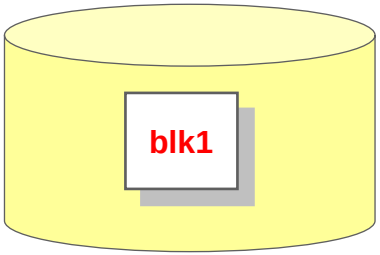


(1) Block 1을buffer list의 buffer를 검색한다.

(2) Block 1의 buffer를 찾았다.



(3) pMem에 저장된 blk 데이터를 BufRead로 전달된 pData에 복사.



Notice: 이미 검색된 **buffer**가 **clean list**에 있기 때문에 **Clean list**에 넣을 필요 없음. 그냥 그대로 두면 됨!!

Buffer cache로 block을 저장하기

- Void BufWrite(int blkno, char* pData)
  - Blkno: 저장할 block 번호
  - pData: 저장할 블록을 담은 메모리 공간
  - 디스크에 블록 저장 동작을 BufWrite가 담당한다.
  - 하지만, DevWriteBlock 흉내만 내고, 실제 디스크 저장은 BufSync가 담당
- 기본 동작
  - 저장할 블록을 Buffer에 저장하고 Dirty list로 이동함. 디스크 저장은 없음.

Disk I/O without Buffer Cache

```
int blkno = 1;
char pData[BLK_SIZE];

DevReadBlock(blkno, pData);
pData[10] = 10; // modify
DevWriteBlock(blkno, pData);
```

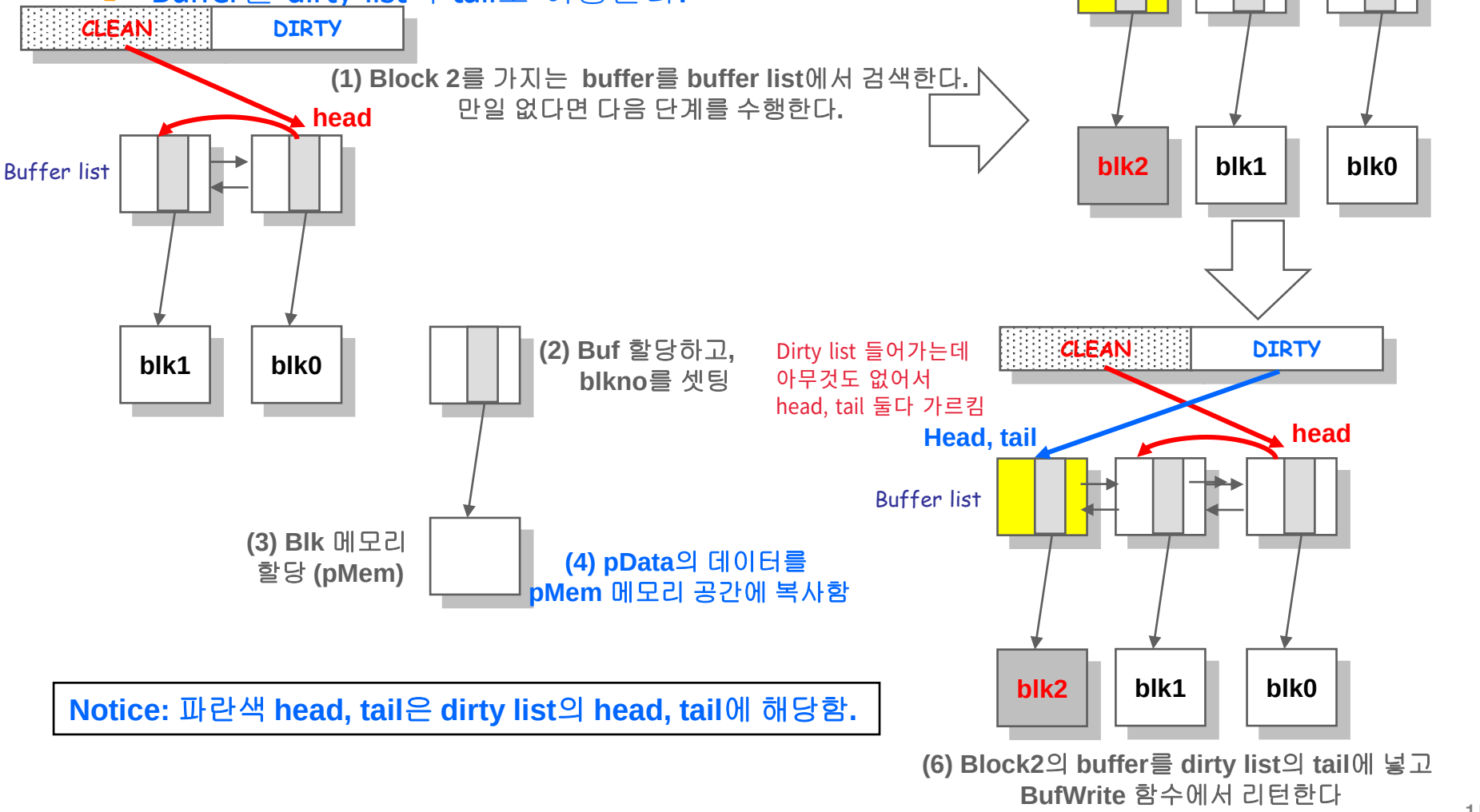
Disk I/O with Buffer Cache

```
int blkno = 1;
char pData[BLK_SIZE];

BufRead(blkno, pData);
pData[10] = 10; // modify
BufWrite(blkno, pData);
BufSync();
```

Buffer cache로 block을 저장하기

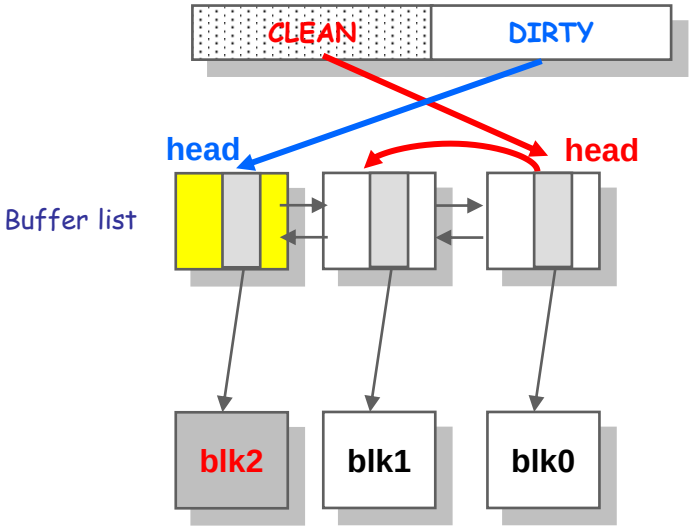
- BufWrite(2, pData)
  - 저장할 block이 buffer cache에 없다고 가정함.
  - 해당 block은 디스크에 저장되지 않은 새로운 block.
  - Buffer는 dirty list의 tail로 이동한다.



Notice: 파란색 head, tail은 dirty list의 head, tail에 해당함.

Buffer cache로 block을 저장하기

- BufWrite(2, pData)
  - 저장할 block이 buffer cache에 있다면? 단, 해당 block이 dirty라고 가정함



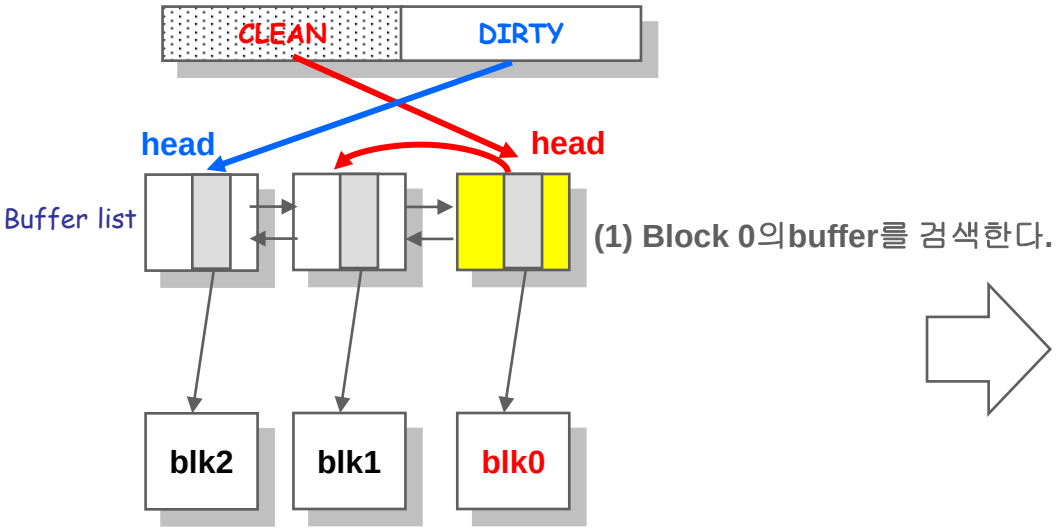
(1) Block 2의 buffer를 buffer list에서 검색한다.

(2) 찾았다. Block 2의 buffer가 가지는 pMem 메모리 공간에 pData 데이터를 복사하고 리턴함

Notice: 이미 검색된 **buffer**가 **dirty list**에 있기 때문에 **dirty list**에 넣을 필요 없음. 그냥 그대로 두면 됨!!

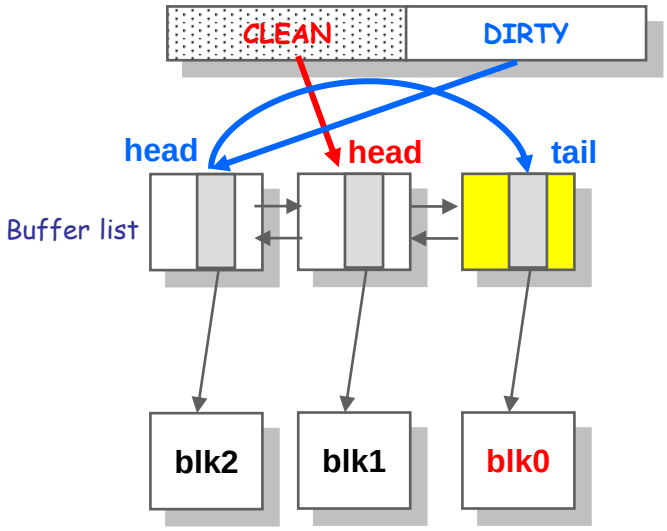
Buffer cache로 block을 저장하기

- BufWrite(0, pData)
  - 저장할 block이 buffer cache에 있다면? 단, 해당 block이 clean라고 가정함
  - Clean list의 buffer는 dirty list의 tail로 이동한다.



(1) Block 0의buffer를 검색한다.

(2) 찾았다. Block 0의 buffer가 가지는 pMem 메모리 공간에 pData 데이터를 복사한다



(3) Block 0의 buffer를 clean list에서 삭제하고, dirty list의 tail로 이동한다

Buffer list 상 위치는 그대로 둔다.
단지, clean list 에서 뺴고 dirty list 의 tail 부분에 넣는다.

Dirty block을 디스크로 저장하기

- Void BufSync(void)
  - Dirty list에 연결된 buffer의 블록들을 디스크로 저장한다. 단, head부터.
  - 저장 후에, buffer는 dirty list에서 clean list의 tail로 이동한다.

Disk I/O without Buffer Cache

```
int blkno = 1;
char pData[BLK_SIZE];

DevReadBlock(blkno, pData);
pData[10] = 10; // modify
DevWriteBlock(blkno, pData);
```

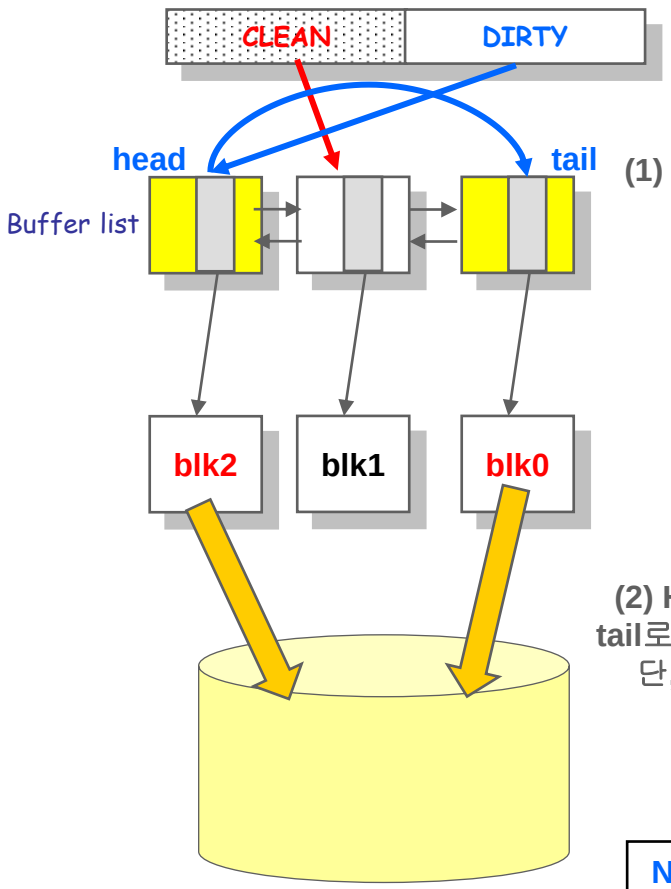
Disk I/O with Buffer Cache

```
int blkno = 1;
char pData[BLK_SIZE];

BufRead(blkno, pData);
pData[10] = 10; // modify
BufWrite(blkno, pData);
BufSync();
```

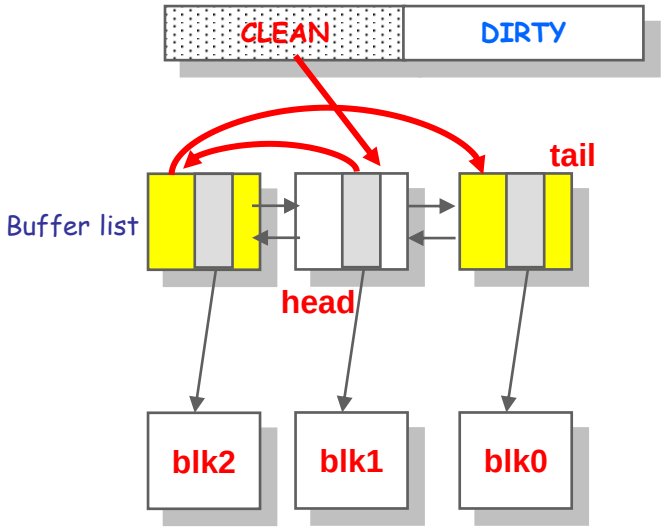
Dirty block을 디스크로 저장하기

Sync 구현!!



(1) Dirty list의 head를 확인함

(2) Head의 buffer의 block부터 tail로 이동하면서 디스크로 저장함. 단, DevWriteBlock을 사용함



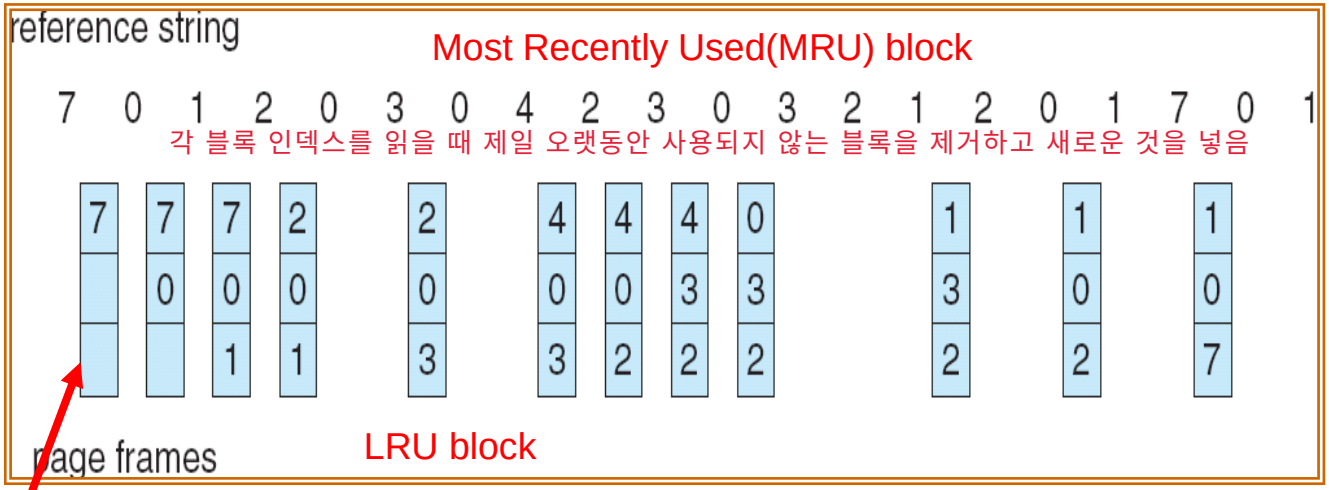
(3) Dirty list의 head에 있는 buffer부터 차례대로 clean list로 이동한다.

Notice: 구현 시, **block 2, 0** 순서로 **block** 단위로 **(2), (3)** 동작을 수행해야 함.
첫째, **block 2**의 **buffe**를 디스크로 저장하고, **clean list**로 이동한다.
둘째, **block 0**의 **buffe**를 디스크로 저장하고, **clean list**로 이동한다.

Buffer Replacement

- LRU (Least Recently Used) block replacement
  - The kernel use the recent past as an approximation of the near future accesses. 가까운 과거에 접근 된 메모리가 가까운 미래에 접근될 가능이 높으며 먼 과거에 접근된 메모리가 먼 미래에 접근될 가능성도 높다.
- Methodology
  - Replace the block that has not been used for the longest period of time.

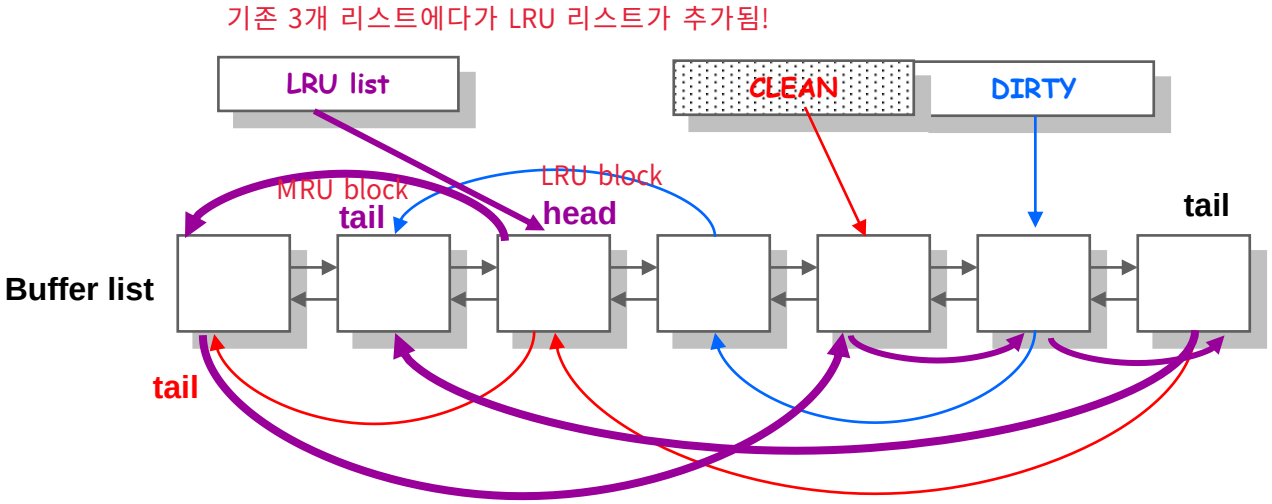
cache hit이 잘 발생하여 효율이 좋아짐



Buffer cache includes 3 buffers

Complete Structure of Buffer Cache

- Buffer replacement가 왜 필요할까?
  - 최대 buffer 개수가 N개라고 가정하자. 왜 N개로 정해 놓을까? 더 많이 할당하고 싶어도 메모리 부족으로 buffer를 무제한 할당할 수 없다.
  - N개가 할당된 후에, 추가로 buffer를 요구할 때 N개 중에 한 개를 선택해서 재사용해야 한다 → Buffer replacement.



Notice
> LRU list의 head는 가장 오랫동안 접근되지 않은 buffer(LRU block)를 가리킴.
> LRU list의 Tail은 가장 최근에 접근한 buffer(MRU block)를 가리킴.

Buffer Replacement

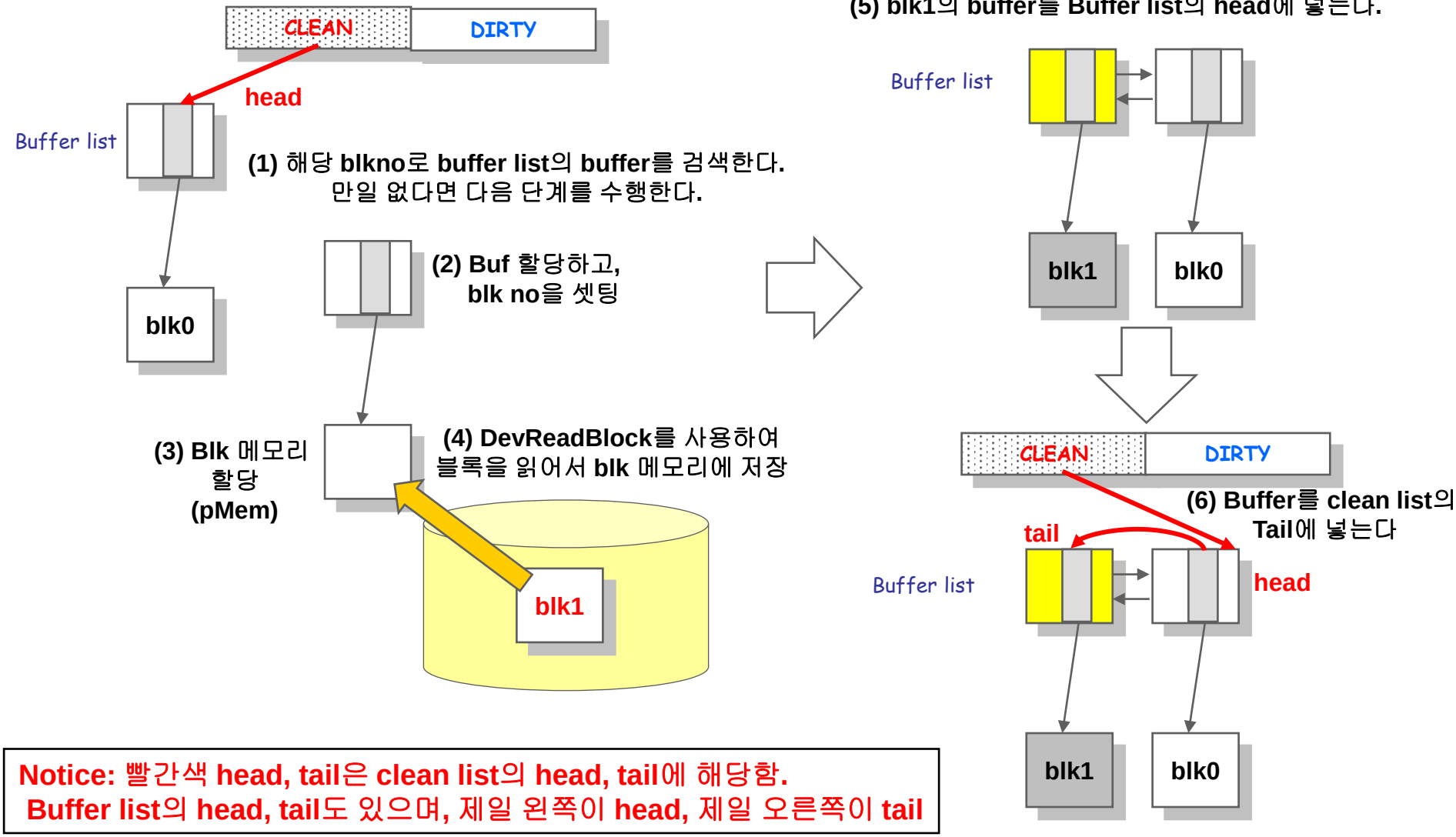
- Buffer replacement
  - 발생 원인: 최대 buffer 개수를 할당한 상황에서, 추가적인 buffer 할당을 원하면 더 이상 buffer를 할당할 수 없다.
  - Replacement 방법: LRU list에서 한 개의 victim buffer를 선택하고, 그 buffer를 재사용함.
  - 선택 방법: LRU list에서 가장 오랫동안 접근되지 않은 buffer를 선택함.
    LRU block을 선택 (LRU list의 head임)

- LRU list 관리
  - 현재 접근되는 buffer는 LRU list의 tail에 넣는다
  - Buffer replacement 시에, LRU buffer인 head를 victim으로 선택하고, 새로운 블록을 위해 재사용함

- Victim 상태에 따른 disk I/O
  - Buffer의 block이 dirty인 경우, 변경된 데이터를 disk에 저장한다. 그렇지 않으면 최신 데이터의 손실이 발생하기 때문. 그후에 buffer를 재사용함.
    flush 되지 않는 buffer cache 상 데이터를 victim 으로 선택되어 새로운 블럭 데이터가 들어올 경우 최신 데이터가 날라가니 문제가 생김
  - Clean이면 disk로 저장하지 않고, buffer를 재사용한다.

Buffer cache에서 block 읽기

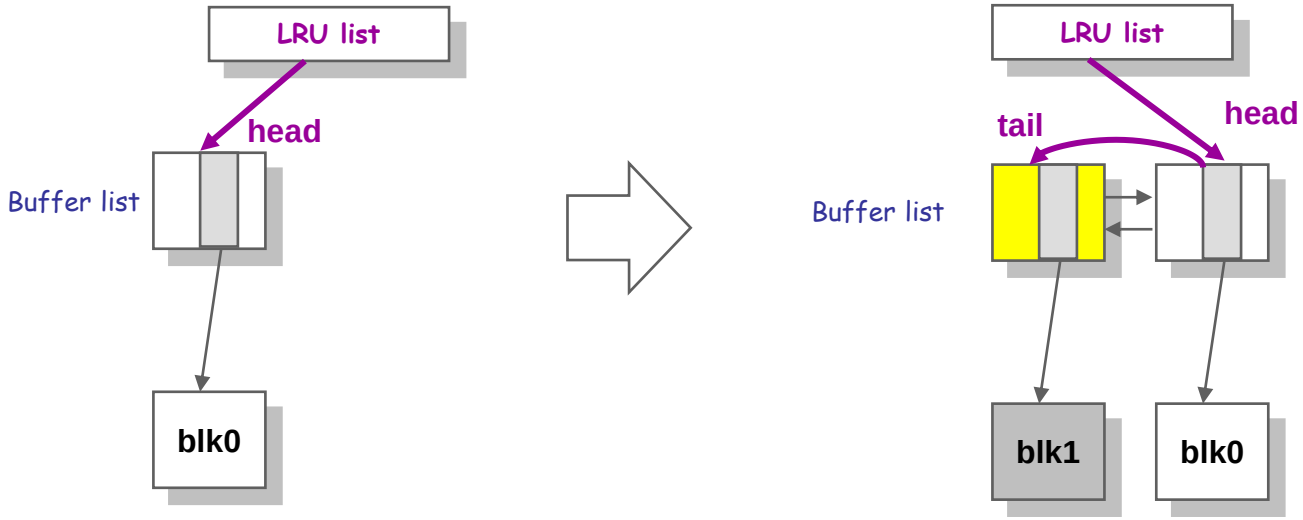
BufRead(1, pData)



Notice: 빨간색 head, tail은 clean list의 head, tail에 해당함.
Buffer list의 head, tail도 있으며, 제일 왼쪽이 head, 제일 오른쪽이 tail

Buffer cache에서 block 읽기

- LRU list 관리
  - Read, Write 관계 없이 현재 접근되는 Buffer는 무조건 LRU list의 Tail로 이동
    읽든 쓰든 접근하는 것이니깐 무조건 LRU 리스트 꼬리에 들어감

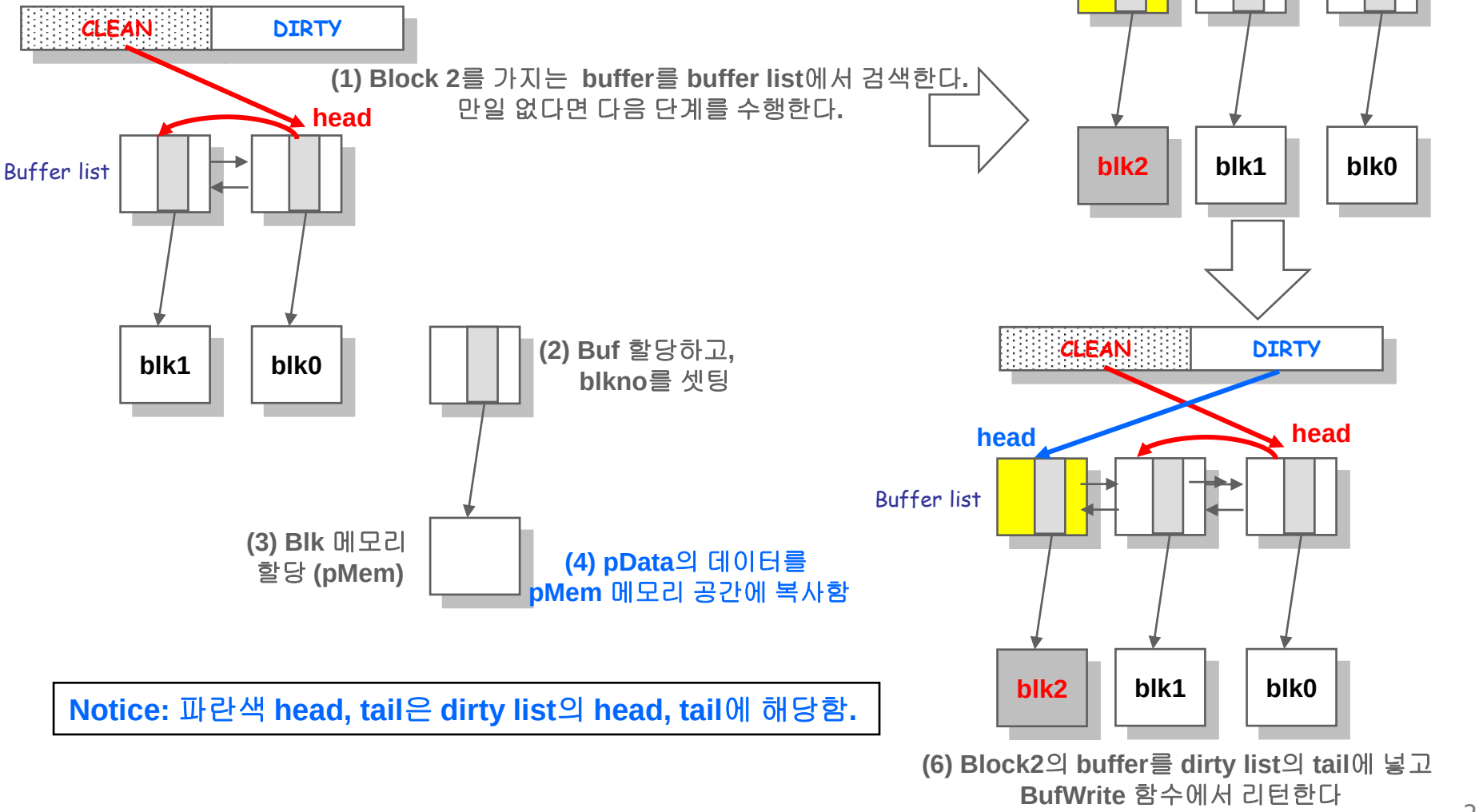


(8) BufRead에서 리턴하기 전에 Buffer를 LRU list의 tail에 넣는다.

Notice
> LRU list의 head는 가장 오랫동안 접근되지 않은 buffer(LRU block)를 가리킴.
> LRU list의 Tail은 가장 최근에 접근한 buffer(MRU block)를 가리킴.

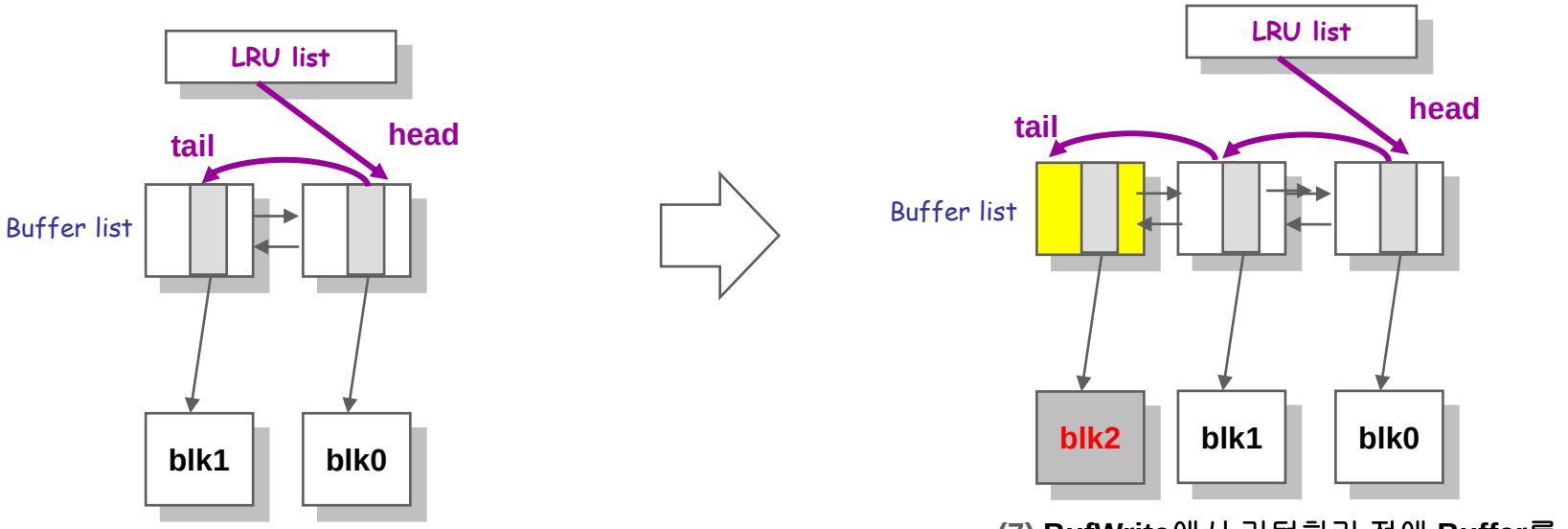
Buffer cache로 block을 저장하기

- BufWrite(2, pData)
  - 저장할 block이 buffer cache에 없다고 가정함.
  - Buffer는 dirty list의 tail로 이동한다.



Notice: 파란색 head, tail은 dirty list의 head, tail에 해당함.

Buffer cache로 block을 저장하기



(7) **BufWrite**에서 리턴하기 전에 **Buffer**를 **LRU list**의 **tail**에 넣는다.

Buffer cache로 block을 저장하기

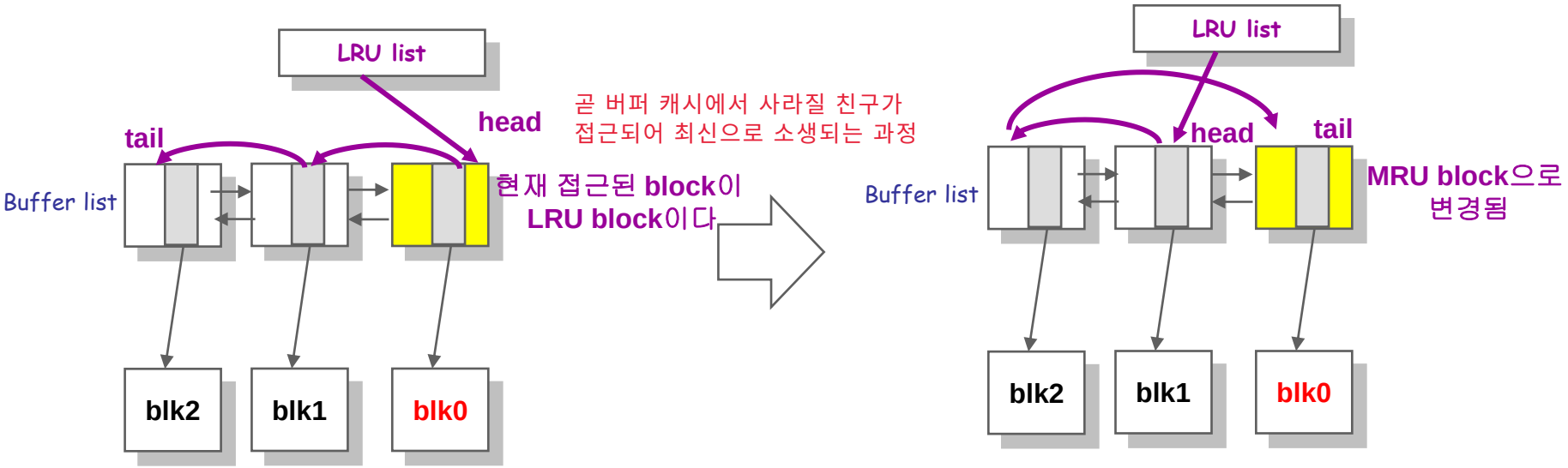
- BufWrite(0, pData)
  - 저장할 block이 buffer cache에 있다면? 단, 해당 block이 clean라고 가정함
  - Clean list의 buffer는 dirty list의 tail로 이동한다.

(2) 찾았다. Block 0의 buffer가 가지는 pMem 메모리 공간에 pData 데이터를 복사한다

(3) Block 0의 buffer를 clean list에서 삭제하고, dirty list의 tail로 이동한다

Buffer cache로 block을 저장하기

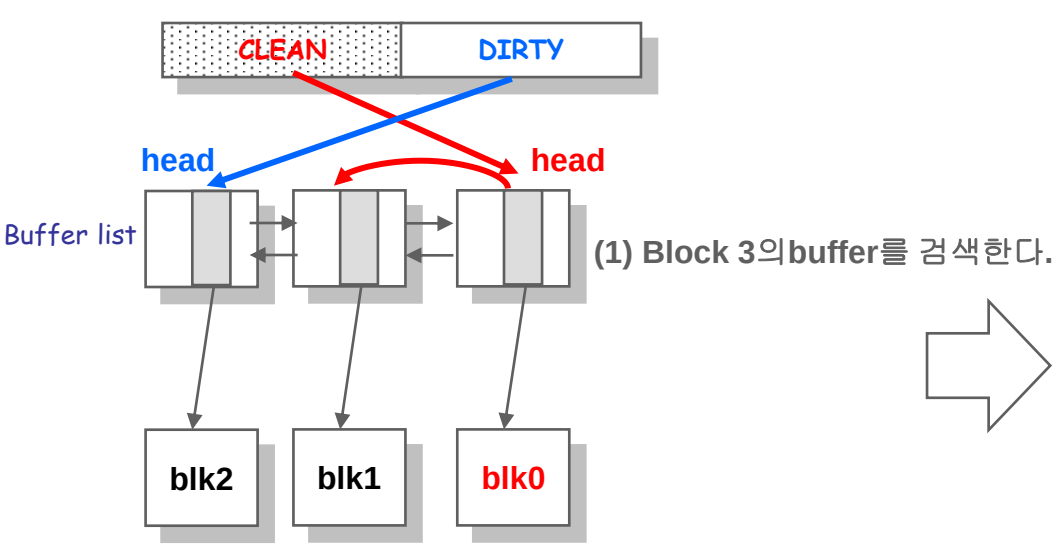
- LRU list 관리
  - Buffer cache에 있는 buffer가 접근되었다면, LRU list의 tail로 이동한다



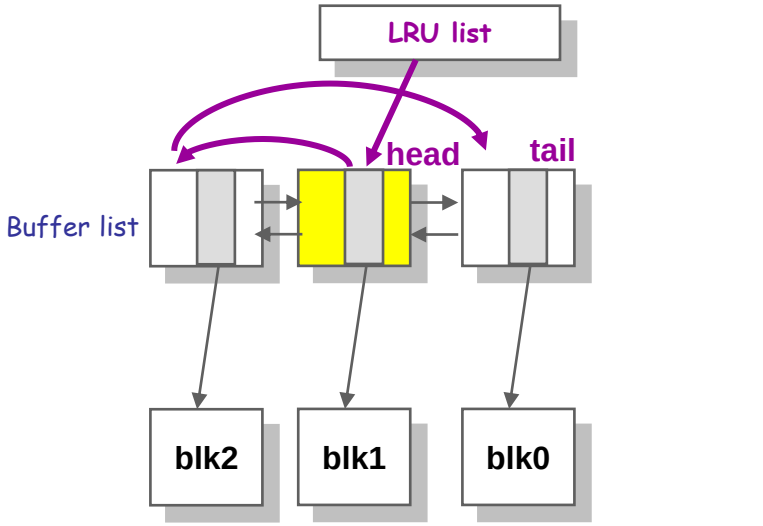
(4) **BufWrite**에서 리턴하기 전에 **Buffer**를 **LRU list**의 **tail**에 넣는다.

Buffer cache로 block을 저장하기 : :Buffer Replacement 발생

- BufRead(3, pData)
  - Buffer cache에 없는 Block 3을 읽는다.
  - 최대 buffer 개수가 3이라면, 가용 buffer가 없음.



(1) Block 3의buffer를 검색한다.

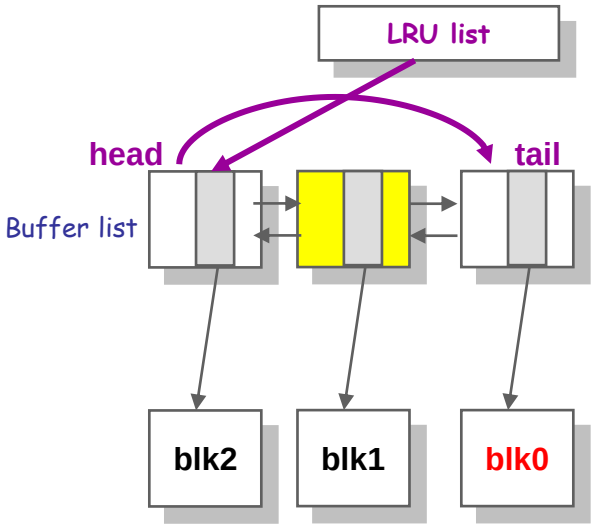


(2) 없다. Block 3의buffer를 할당하려 하지만, 가용할 수 있는 buffer가 없다.

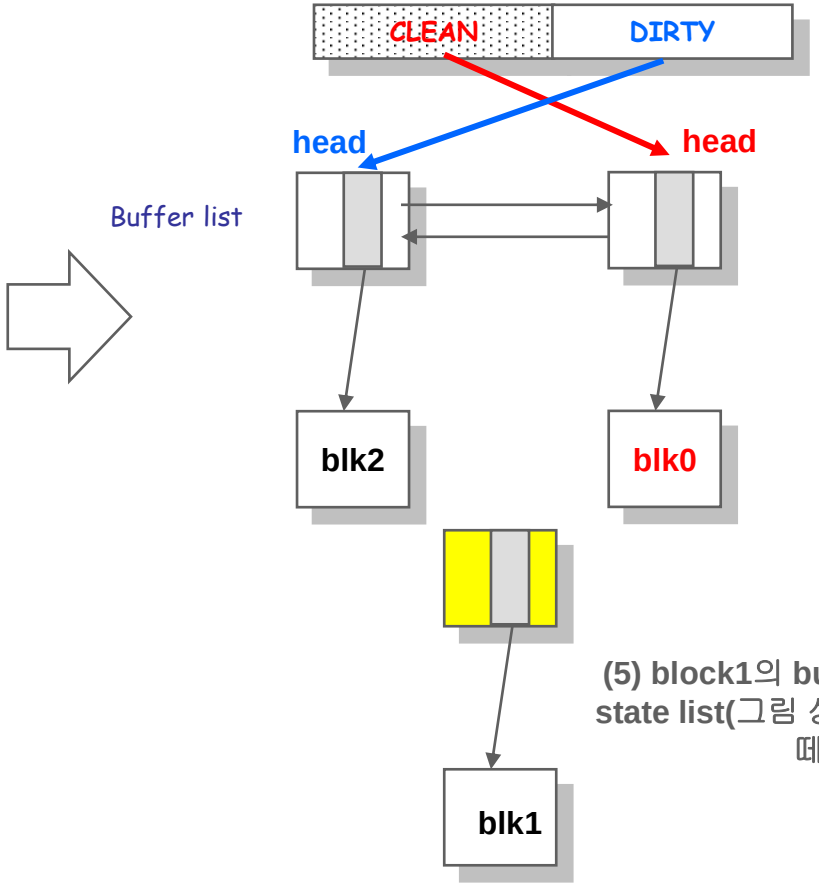
(3) LRU list를 확인하고, LRU buffer(head)를 Victim으로 선택한다. **Block 1의 buffer가 victim이다.**

Buffer cache로 block을 저장하기

- Victim을 LRU, state List (clean 또는 dirty list), buffer list에서 떼어낸다
  - 떼어낸 buffer는 새로운 block을 저장하기 위해 사용됨.



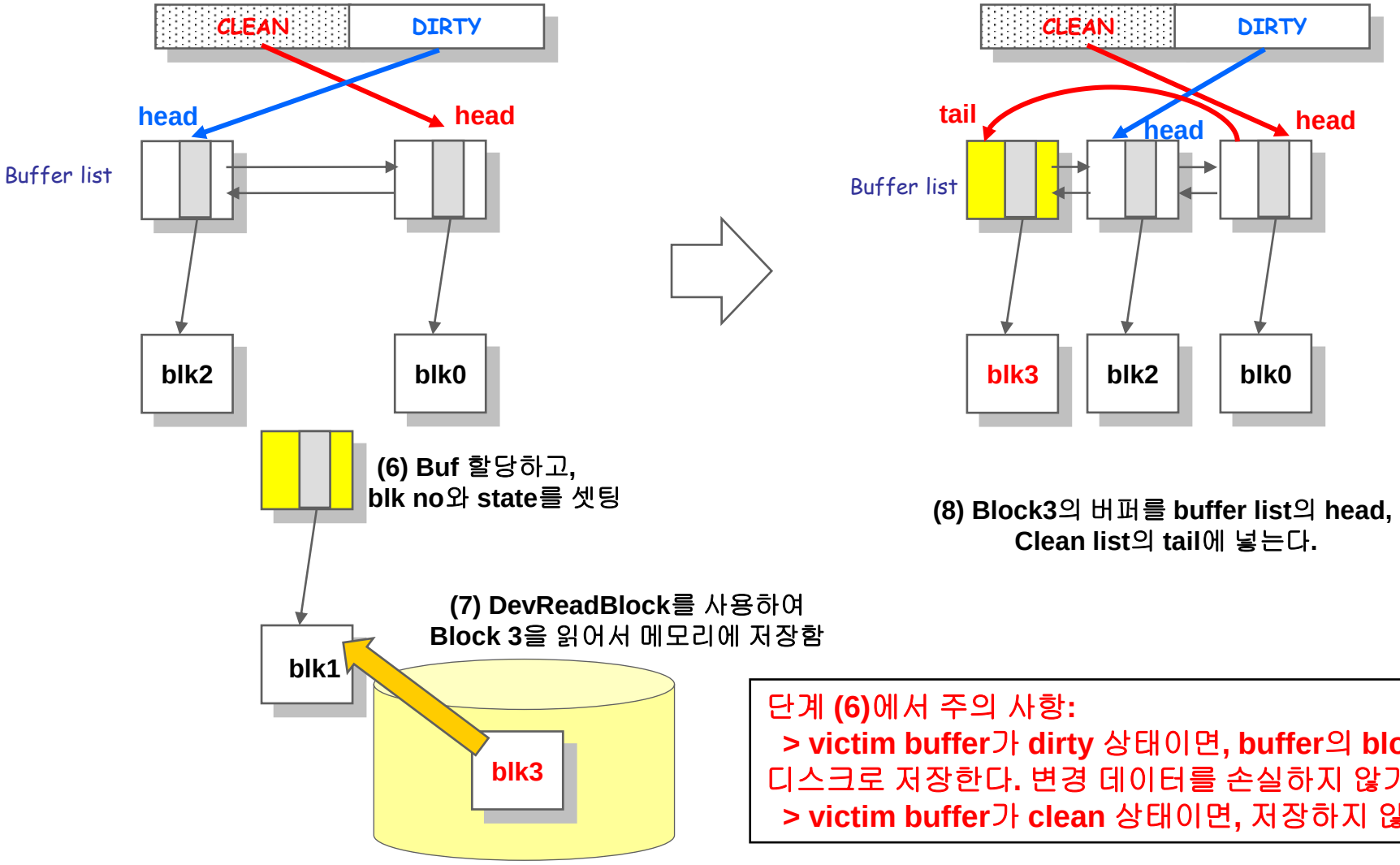
(4) LRU list의 head에서
block 1의 buffer를 떼어낸다



(5) block1의 buffer를 buffer list와
state list(그림 상에서 dirty list)에서
떼어낸다

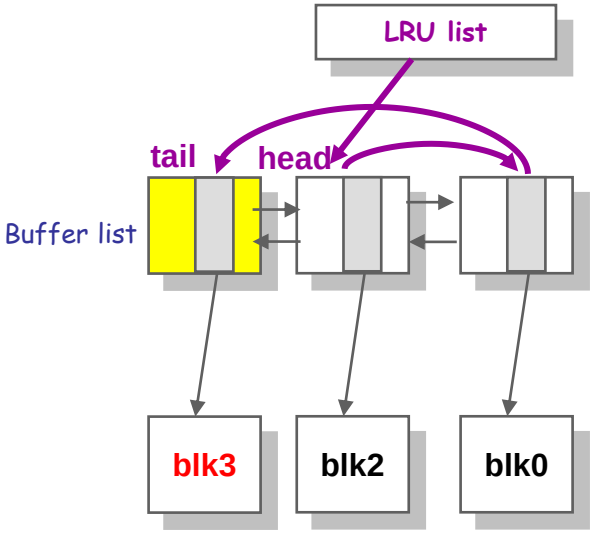
Buffer cache로 block을 저장하기

- Victim buffer의 재사용
  - 새로운 buffer 할당 후의 동작과 동일함. Buffer, clean 또는 dirty, LRU list에서

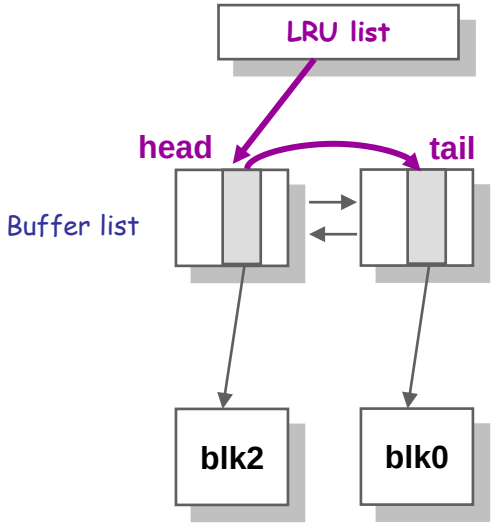


단계 **(6)**에서 주의 사항:
> victim buffer가 **dirty** 상태이면, **buffer**의 **block**을 디스크로 저장한다. 변경 데이터를 손실하지 않기 위함.
> victim buffer가 **clean** 상태이면, 저장하지 않음.

Buffer cache로 block을 저장하기



(9) Block 3의 buffer를 LRU list의 tail에 넣는다.



단계 **(4)**에서 **victim**이 제거된 후의 **LRU list**의 상태

유의 사항

- 구현 해야할 함수들
  - BufInit, BufRead, BufWrite, BufSync
- 구현 해야할 파일들
  - buf.c 구현. buf.c에 임의의 헤더 파일(예, temp.h 등)을 include 가능
  - disk.c, disk.h 제공됨. 수정 불가(수정하면 0점 처리)
  - buf.h에는 구현 해야할 function prototype 선언함.
  - main.c에서 buf.h를 include해서 testcase가 제공되기 전에 각자 테스트함
  - 향후 testcase를 포함하는 main.c을 제공할 계획

main.c

```
int blkno = 1;
char pData[BLK_SIZE];

BufRead(blkno, pData);
pData[10] = 10; // modify
BufWrite(blkno, pData);
BufSync();
```

buf.c

```
#include "buf.h"

Void BufInit(void)
{
    ...
}
Void BufRead(int blkno, char* pData)
{
   ... // implement this func.
}
...
```

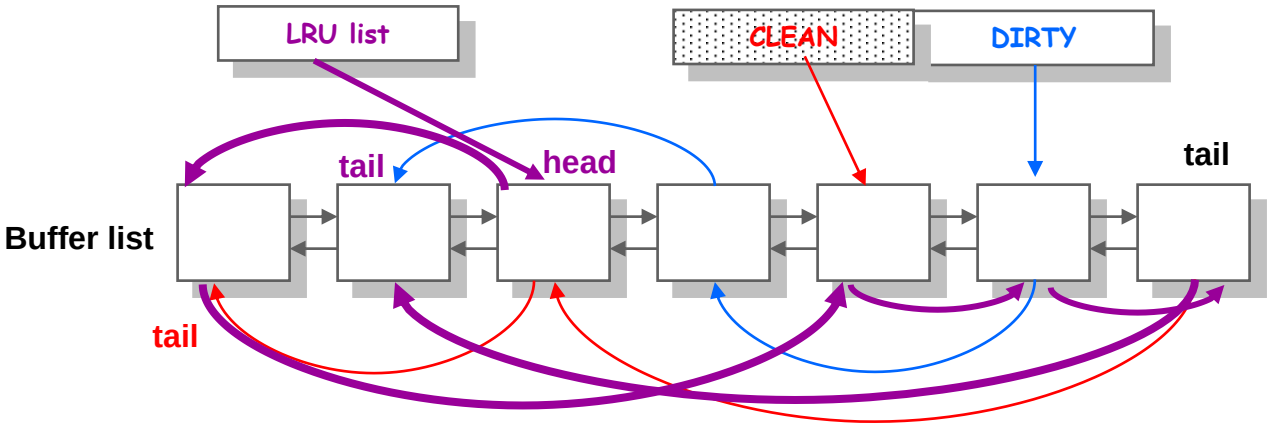
간단한 TestCase 시나리오

main.c

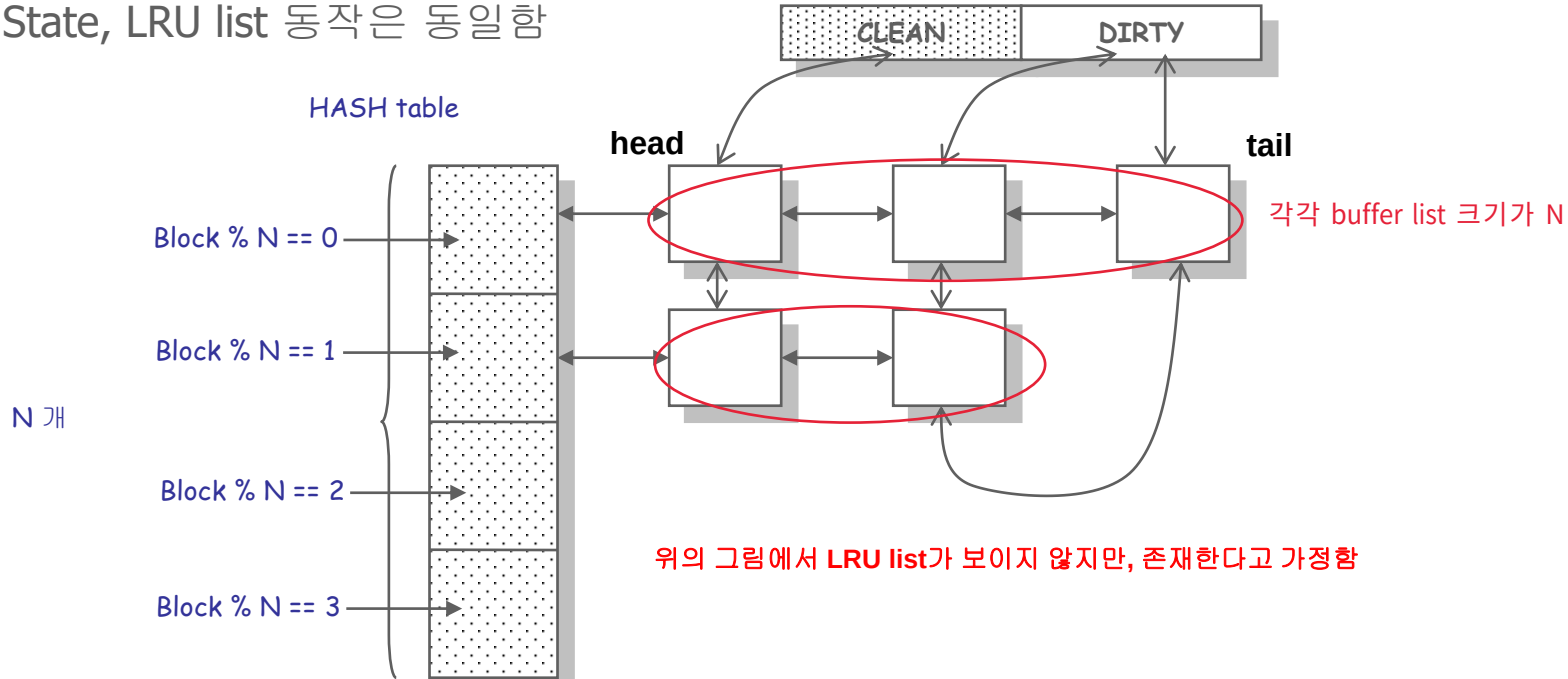
```
Char pData[BLK_SIZE];
BufInit();
// pData 메모리에 초기화
BufWrite(0, pData); // 0번 블록의 버퍼는 dirty 상태이며, dirty list에 있어야 함
BufWrite(1, pData); // 1번 블록의 버퍼는 dirty 상태이며, dirty list에 있어야 함
BufWrite(2, pData); // 2번 블록의 버퍼는 dirty 상태이며, dirty list에 있어야 함
BufWrite(3, pData); // 최대 buf가 3개라면 0번 블록의 버퍼가 교체. Dirty인 0번은 디스크 저장
BufWrite(4, pData); // 1번 블록의 버퍼가 교체가 발생. Dirty이므로 디스크로 저장
BufWrite(5, pData); // 2번 블록의 버퍼가 교체가 발생. Dirty이므로 디스크로 저장
BufWrite(6, pData); // 3번 블록의 버퍼가 교체가 발생. Dirty이므로 디스크로 저장
// -- blk no 0부터 3까지는 이미 디스크에 저장되어 있어야함. 4~6는 디스크에
// -- 저장되어 있지 않고 대신 dirty 상태로 메모리에 존재함.
printLRUList(); printStateList(); printBufList();

// 다음부터 디스크에 저장되었지만, buf cache에 없는 블록을 읽음
BufRead(0, pData); // 4번 블록이 디스크로 저장되고 교체됨, 0번은 clean list로.
BufRead(1, pData); // 5번 블록이 디스크로 저장되고 교체됨, 1번은 clean list로.
BufRead(2, pData); // 6번 블록이 디스크로 저장되고 교체됨, 2번은 clean list로.
printLRUList(); printStateList(); printBufList();
```

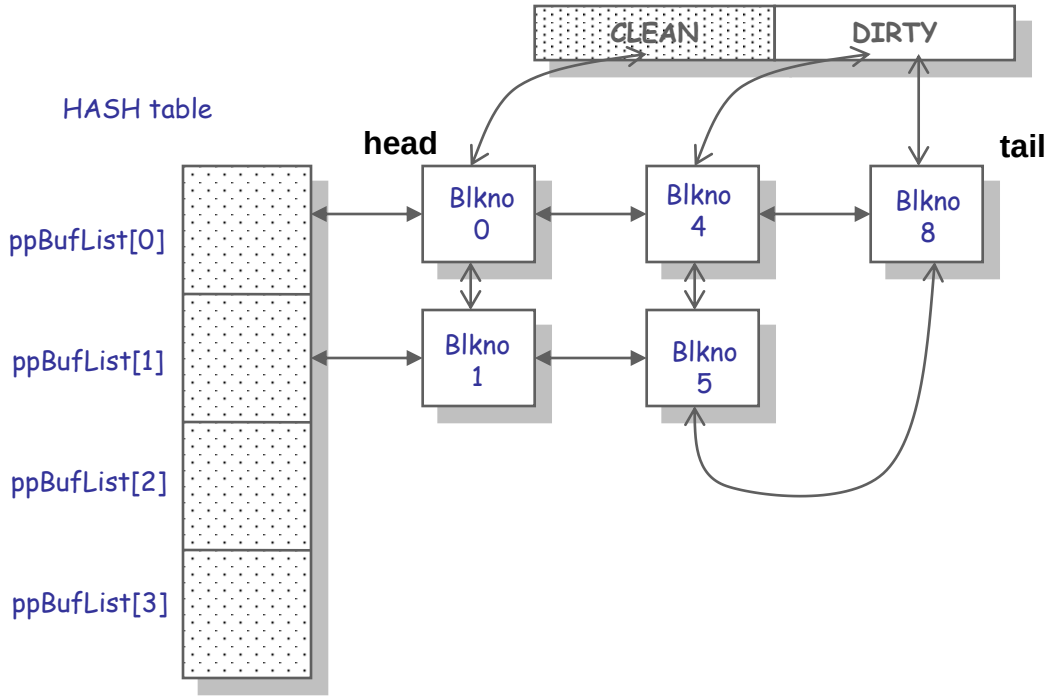
Buffer list의 Hash Table 관리



- 기준 Buffer list를 Hash Table 기반의 Buffer list로 구현.
- 새로운 buffer list는 N개로 구성된 buffer list로 구성됨. N = hash table entry 개수
- blkno % N = entry num. 해당 블록은 entry num의 buffer list의 head에 들어간다.
- State, LRU list 동작은 동일함



Buffer list의 Hash Table 관리



#define HASH_ENTRY_NUM (4)

TAILQ 사용하는 버전
TAILQ_HEAD(bufList, Buf) ppBufList[HASH_ENTRY_NUM];

TAILQ 사용하지 않는 버전
Buf* pBufList[HASH_ENTRY_NUM];

void GetBufInfoInBufferList(Buf** ppBufInfo, int* pNumBuf);

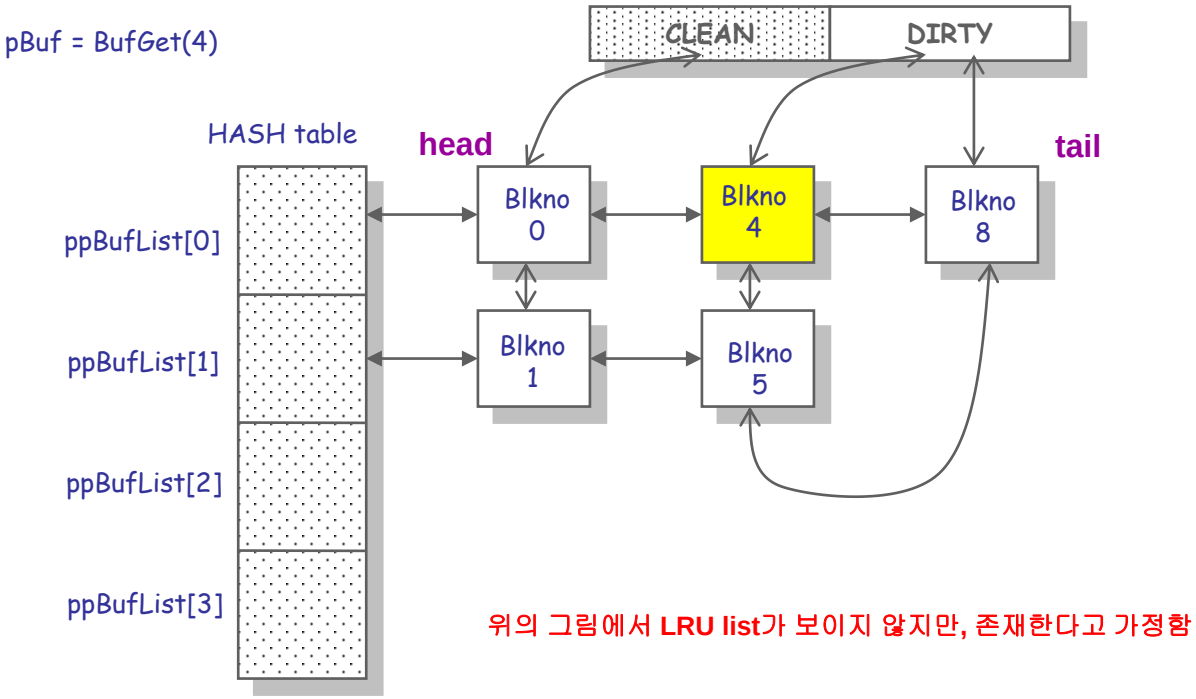
↓

void GetBufInfoInBufferList(int hashEntNum, Buf** ppBufInfo, int* pNumBuf);

hashEntNum 번에 해당하는 해쉬 테이블의 버퍼 리스트를 가져옴

Buffer Cache에서 Buffer 검색

- Buf* BufGet(int blkno)
  - blkno의 블록을 가지는 버퍼를 검색해서 해당 버퍼의 주소 값을 반환
  - 해당 버퍼가 없으면 NULL을 반환
  - 유의사항: 검색 후 주소 값만 반환할 뿐이지 Buffer, State, LRU list에서 어떠한 변화(순서, 삭제 등)도 없어야 함.



특정 블록의 버퍼에 대한 **Sync**

- void BufSyncBlock(int blkno) clean block 에 대한 예외처리 안해도됨
  - blkno의 블록을 가지는 버퍼가 dirty block이라고 가정함.
  - 해당 버퍼를 디스크로 Sync 또는 저장함.
  - 유의사항: BufSync()와 동일한 동작이지만, 특정 블록만 디스크로 Sync함.
  - 해당 Buffer가 Sync되면, dirty list → clean list, dirty state → clean state

```
Buf* pBuf;

pBuf = BufGet(4);
if (pBuf != NULL)
   BufSyncBlock(4);
```