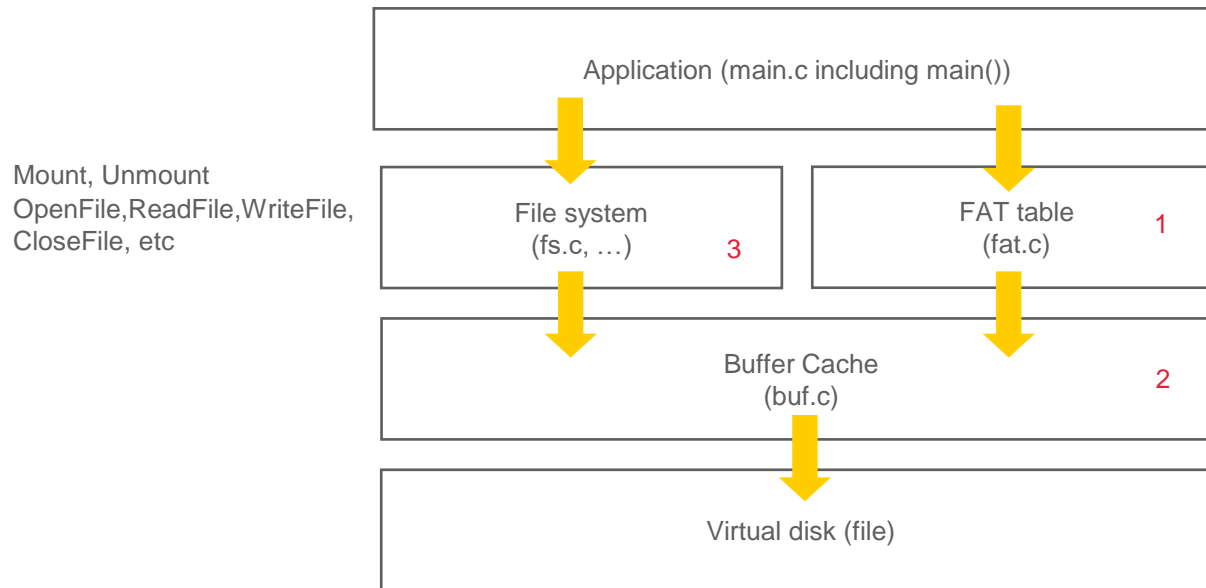


Implementing FAT File System

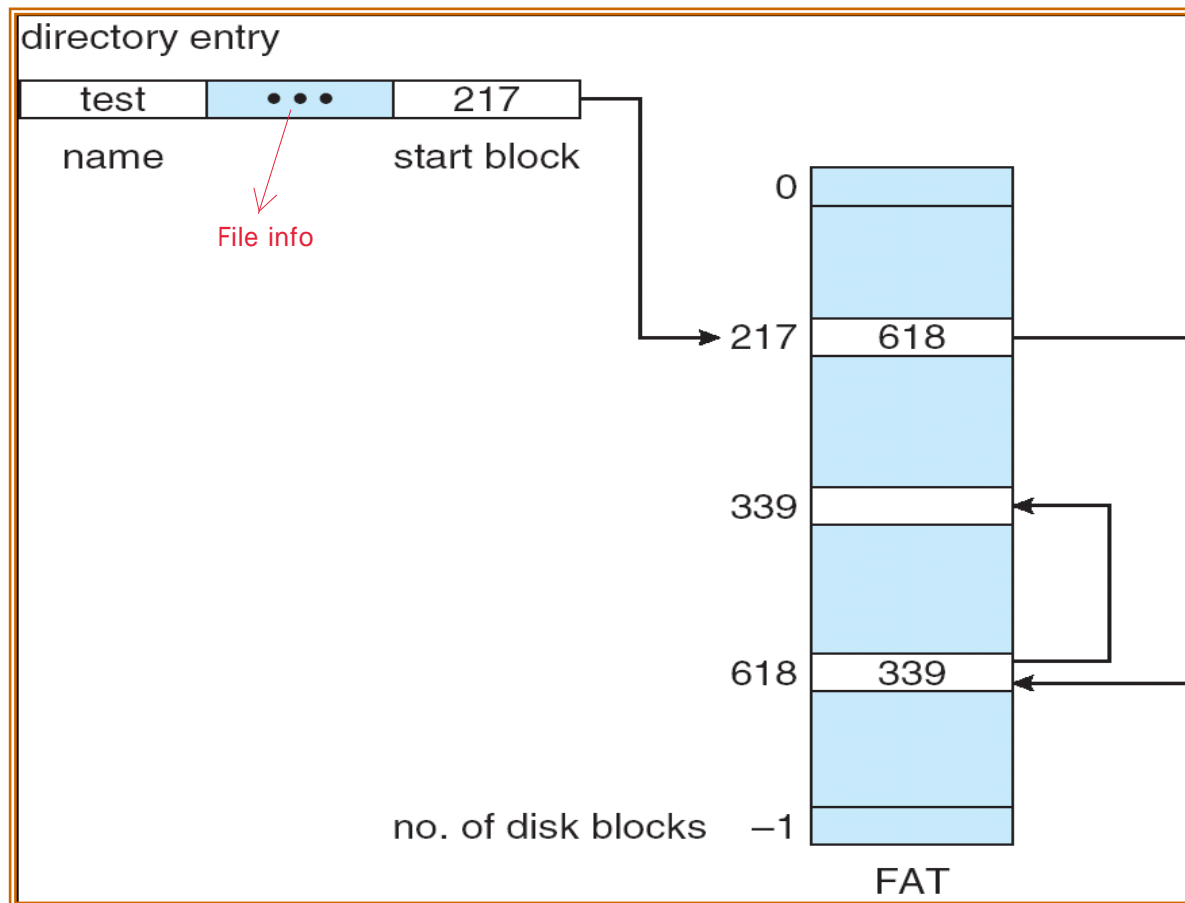
Woo Hyun Ahn (whahn@kw.ac.kr)

Layed Architecture



Linked Allocation - FAT

- File Allocation Table (FAT)
 - An important variation on linked allocation.
 - MS-DOS and embedded systems such as mp3 player and memory stick.
 - A section of disk at the beginning of each volume contains the table.



File system layout

File system Info 또는 superblock

- > File system 전체의 정보를 저장함 (Homework3.doc 참고)
- > 한 개의 블록에 할당

FAT table

- > Homework1에서 구현한 FatInit, FatAdd, FatRemove, FatGetBlockNum 함수를 통해 관리됨

Data Region

- > directory, file data를 저장하는 영역



- 가
-
-

File System Info

```
typedef struct _FileSysInfo {  
    int blocks;           // 디스크에 저장된 전체 블록 개수  
    int rootFatEntryNum;  // 루트 디렉토리의 1st Fat entry 번호  
    int diskCapacity;    // 디스크 용량 (Byte 단위)  
    int numAllocBlocks;   // 파일 또는 디렉토리에 할당된 블록 개수  
    int numFreeBlocks;    // 할당되지 않은 블록 개수  
    int numAllocFiles;    // 전체 파일 및 디렉토리 개수  
    int fatTableStart;    // Fat table의 시작 블록 번호  
    int dataStart;        // data 영역의 시작 블록 번호  
} FileSysInfo;
```

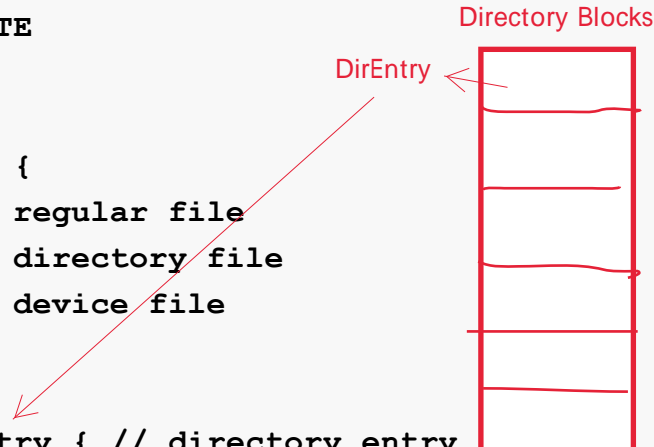
()

Directory Entry

```
typedef enum __AccessMode {
    ACCESS_MODE_READONLY,
    ACCESS_MODE_WRITEONLY,
    ACCESS_MODE_READWRITE
} AccessMode;

typedef enum __FileType {
    FILE_TYPE_FILE,    // regular file
    FILE_TYPE_DIR,     // directory file
    FILE_TYPE_DEV      // device file
} FileType;

typedef struct __DirEntry { // directory entry
    Char name[16];        // file name
    AccessMode mode;      // file access mode: read-only, write-only, read-write
    int startFatEntry;    // file의 1st fat entry 번호
    FileType filetype;    // file type
    int numBlocks;        // file을 구성하는 블록 개수
} DirEntry
```



Directory Blocks

DirEntry

File system layout in detail

Block 크기: 64 bytes

Directory 당 directory entry 개수: 4개 (64/16)

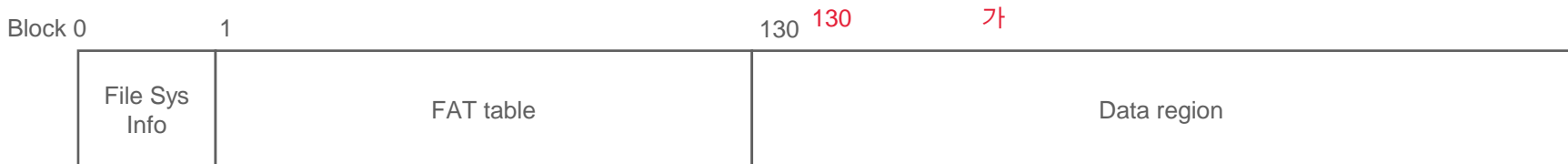
가 16bytes

FAT table 크기: 128 blocks

FAT table에서 사용되지 않는 entry 영역

> Data 영역이 시작되는 블록부터 FAT entry를 할당

> FAT entry 0 ~ 129는 사용되지 않음



유의 사항 super block 1 FAT table 129 Data region

> FatInit, FatAdd, FatRemove, FatGetBlockNum가 BufRead, BufWrite를 호출하도록 수정한다.

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

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



Hw1
Hw2

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

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

Mount 동작

■ Void Format()

- 파일 시스템을 포맷하고 초기화하는 동작을 수행한다.
- 가상 디스크를 초기화하고 **file system info**를 초기화한다.

■ Void Mount()

- 파일 시스템을 포맷이 아닌, 전원을 끄기 전의 파일시스템을 사용하는 동작이다.
- 포맷 대신 가상 디스크를 **open**하는 동작만 수행한다.

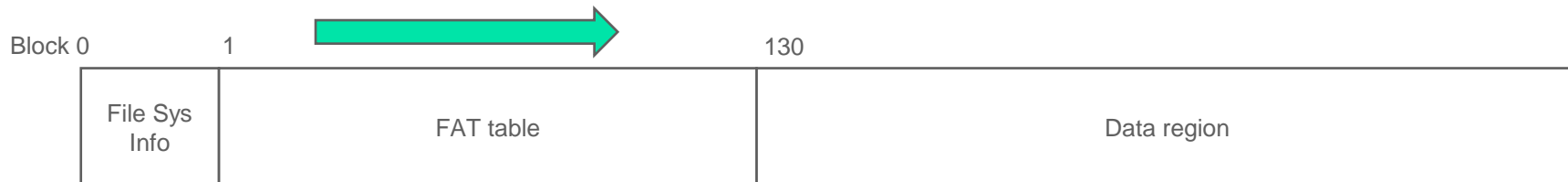
■ Unmount()

- 전원을 끌 때 호출되는 함수라고 간주한다.
- Buffer cache에 저장된 **dirty block**들을 디스크로 저장한 후에, 가상 디스크를 **close**한다. 이때, **BufSync()** 사용한다.

Format()

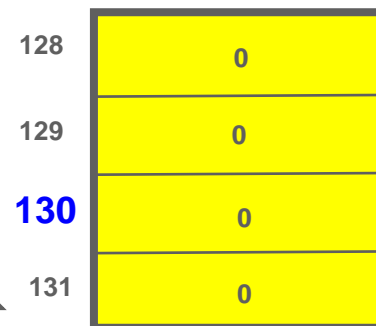
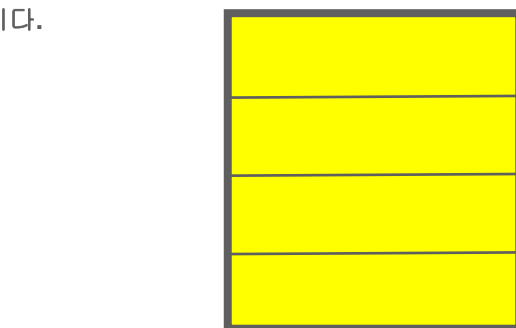
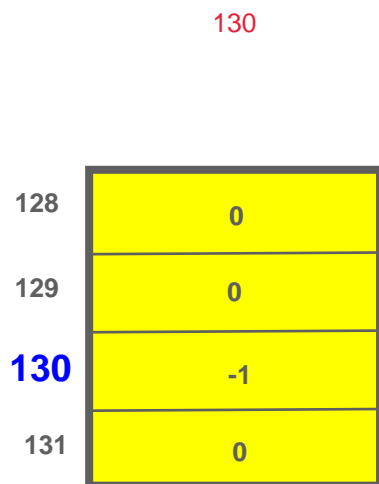
(0) 파일시스템 초기화 단계기 때문에 `FatInit()`을 통해 `Block1`부터 시작하여 `FAT table` 전체를 초기화를 해야 한다.

(1) `FAT table`의 130번 entry부터 빈 entry를 검색함. 검색된 entry 번호가 130이라고 가정함. 검색 및 할당 단계는 (a)~(e)

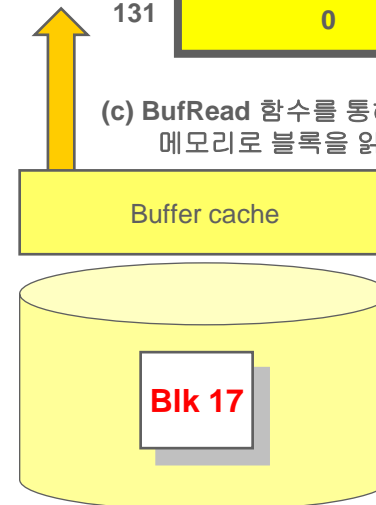


(a) Entry 130이 포함된 FAT 블록을 확인한다.
130은 FAT 시작 블록에서 16번째 블록이다.
블록 번호: $1+16 = 17$ 번

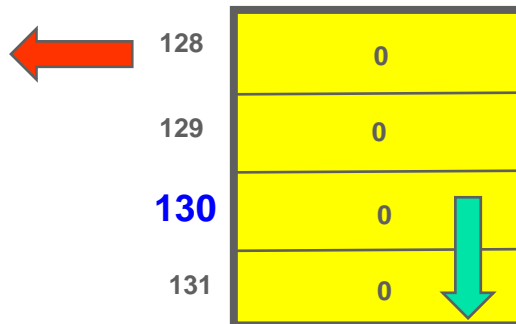
(b) Block 크기의 메모리를 할당한다.



(c) `BufRead` 함수를 통해 할당된 메모리로 블록을 읽는다



(e) 디렉토리 블록을 할당해야기 때문에 Entry를 -1로 변경하고, `BufWrite` 함수를 통해 디스크로 저장함

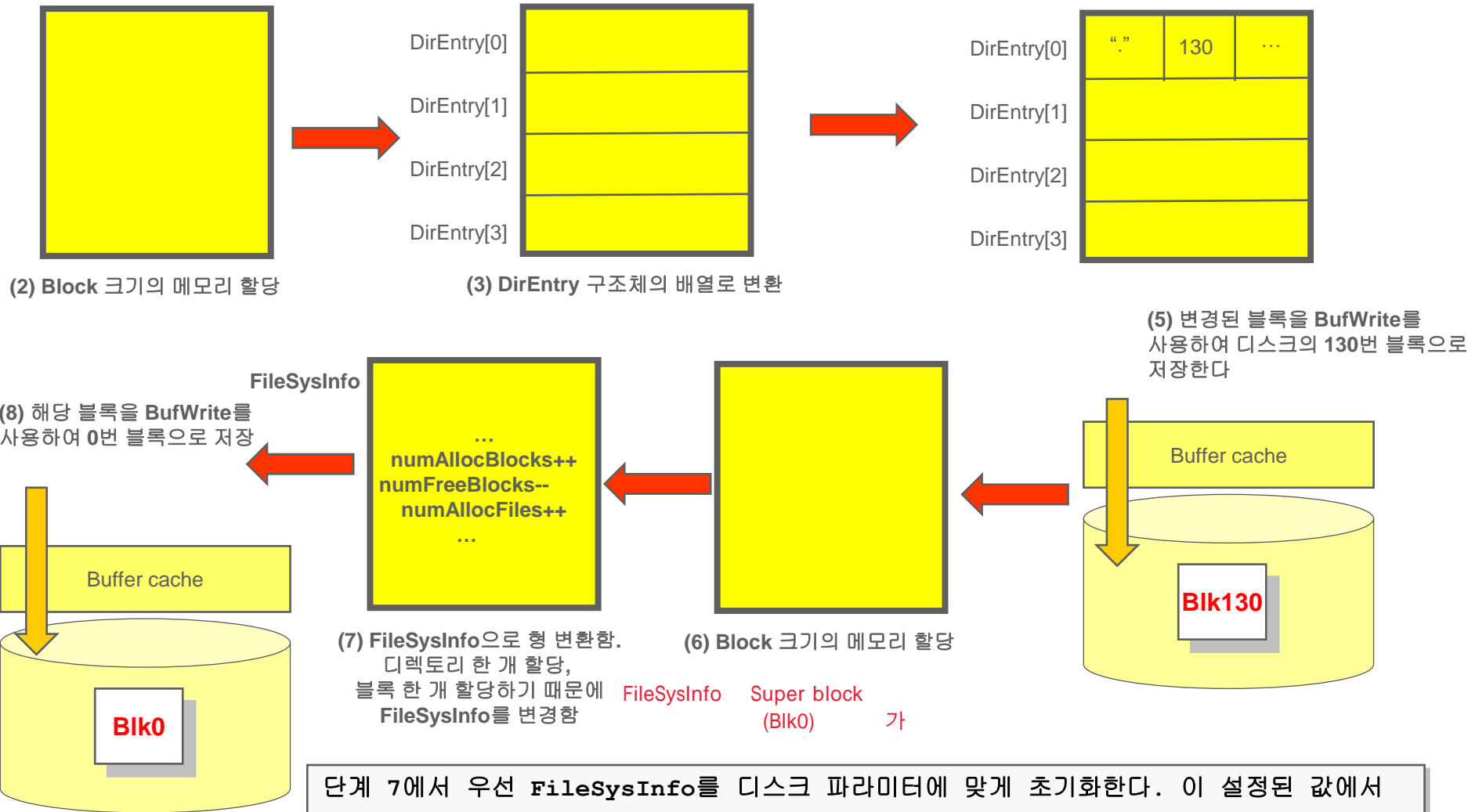


(d) Entry 130부터 빈 entry를 검색한다.
130을 찾았다

Format()

Root 디렉토리 생성 과정

(4) DirEntry[0]의 변수들을 설정함.
> name: "."
> Mode: **ACCESS_MODE_READWRITE**
> startFatEntry: 130
> filetype: FILE_TYPE_DIR
> numBlocks: 1

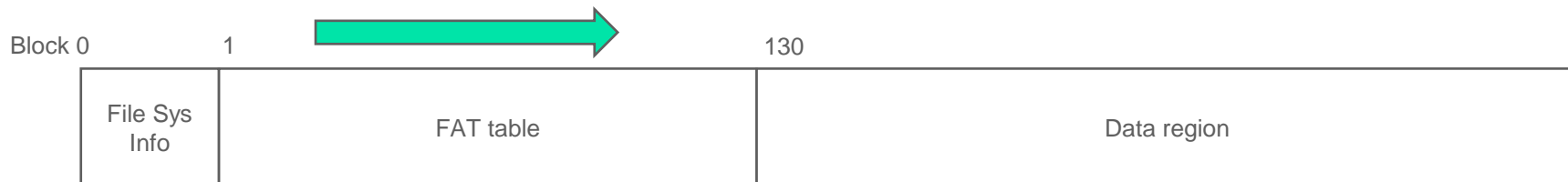


단계 7에서 우선 FileSysInfo를 디스크 파라미터에 맞게 초기화한다. 이 설정된 값에서 numAllocBlocks++, numFreeBlocks--, numAllocFiles++ 후에 디스크로 저장한다.

Directory 생성하기

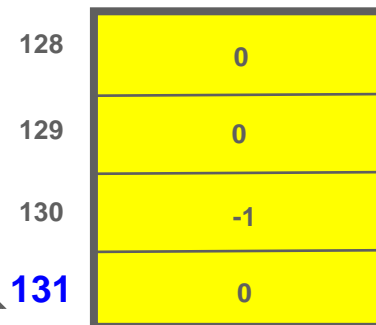
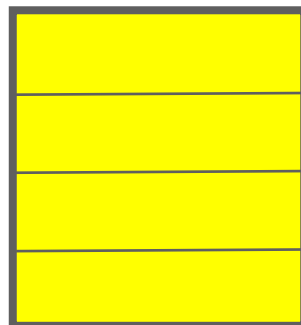
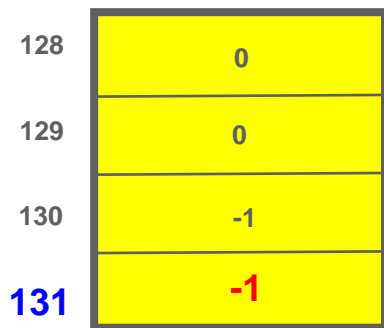
MakeDir("wtmp", mode): tmp에 필요한 디렉토리 블록을 할당

(1) FAT table의 130번 entry부터 빈 entry를 검색함. 검색된 entry 번호가 131이라고 가정함. 검색 및 할당 단계는 (a)~(e)

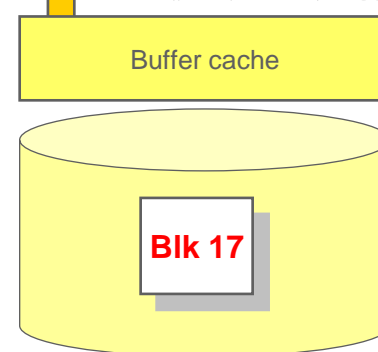


(a) Entry 130이 포함된 FAT 블록을 확인한다.
130은 16번째 FAT 시작 블록에서 16번째 블록이다.
블록 번호: $1+16 = 17$ 번

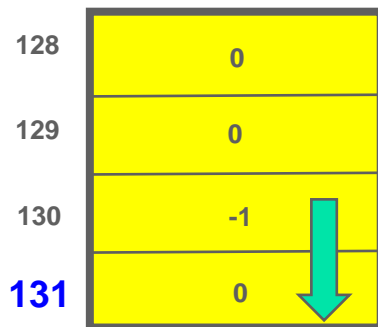
(b) Block 크기의 메모리를 할당한다.



(c) BufRead 함수를 통해 할당된 메모리로 블록을 읽는다



(e) 디렉토리 블록을 할당해야기 때문에
Entry를 -1로 변경하고,
BufWrite 함수를 통해 디스크로 block17을 저장함

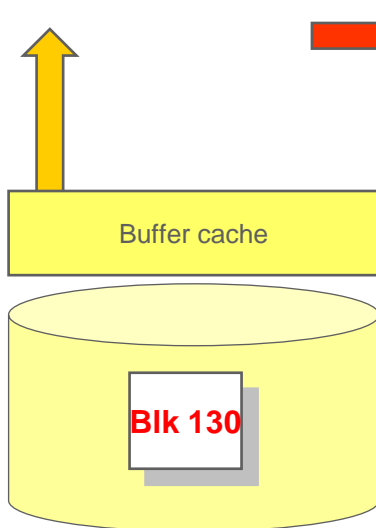


Directory 생성하기

MakeDir("□tmp"): "/" 디렉토리에 tmp를 위한 directory entry를 추가하자

(2) Root 디렉토리의 시작 블록을 확인한다.
1st FAT entry가 130번 블록이다.

(3) 블록 크기의 메모리를 할당하고,
그 메모리로 BufRead 함수를 통해
block 130을 읽는다



DirEntry[0]	"."	130	...
DirEntry[1]			
DirEntry[2]			
DirEntry[3]			

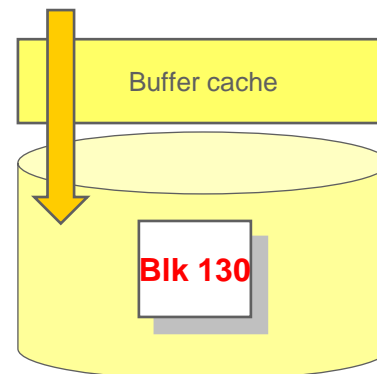
(4) DirEntry로 형 변환하고,
빈 DirEntry를 검색한다.
찾았다. DirEntry[1] 비워있다.

(5) DirEntry[1]의 변수들을 설정함.
> name: "tmp"
read or write > Mode: **mode 값을 저장**
> startFatEntry: **131**
> filetype: FILE_TYPE_DIR
> numBlocks: 1

단계 (1)에서 block 131을 할당 받았다.
이 블록을 tmp 디렉토리의 블록으로 사용한다.

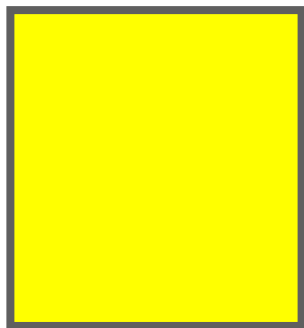
DirEntry[0]	"."	130	...
DirEntry[1]	"tmp"	131	...
DirEntry[2]			
DirEntry[3]			

(6) 변경된 블록을 BufWrite를
사용하여 디스크의 130번 블록으로
저장한다



Directory 생성하기

MakeDir("□tmp"): "tmp" 디렉토리 블록을 초기화하자



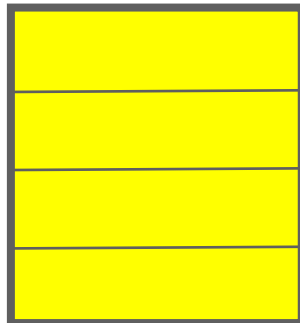
(7) Block 크기의 메모리 할당

DirEntry[0]

DirEntry[1]

DirEntry[2]

DirEntry[3]



(8) DirEntry 구조체의 배열로 변환

(9) DirEntry[0]의 변수들을 설정함.

- > name: "."
- > Mode: 임의의 값
- > startFatEntry: **131**
- > filetype: FILE_TYPE_DIR
- > numBlocks: 1

DirEntry[0]

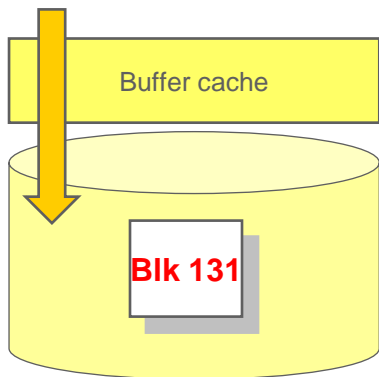
DirEntry[1]

DirEntry[2]

DirEntry[3]

DirEntry[0]	."	131	...
DirEntry[1]			
DirEntry[2]			
DirEntry[3]			

(10) 변경된 블록을 BufWrite를 사용하여 디스크의 131번 블록으로 저장한다. Block 131이 tmp의 directory block이다.



(9) DirEntry[1]의 변수들을 설정함.

- '/' > name: **".."**
- > Mode: 임의의 값
- > startFatEntry: **130**
- > filetype: FILE_TYPE_DIR
- > numBlocks: 1

DirEntry[0]

DirEntry[1]

DirEntry[2]

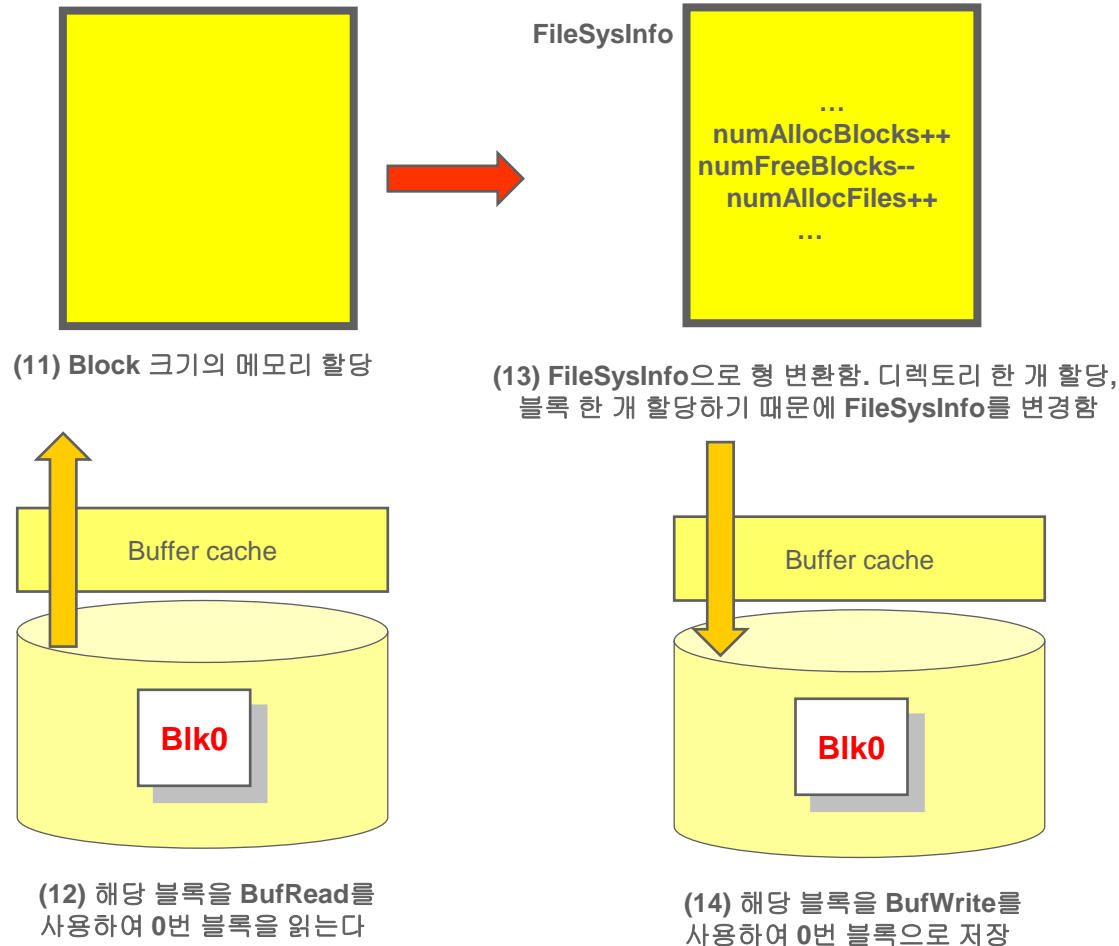
DirEntry[3]

DirEntry[0]	."	131	...
DirEntry[1]	".."	130	...
DirEntry[2]			
DirEntry[3]			

Directory 생성하기

MakeDir("\0tmp"):

- > 이대로 끝나치면 안된다. File system 상태가 변경되었기 때문에 FileSysInfo를 업데이트한다.



파일 생성

`int OpenFile(char* szFileName, OpenFlag flag)`

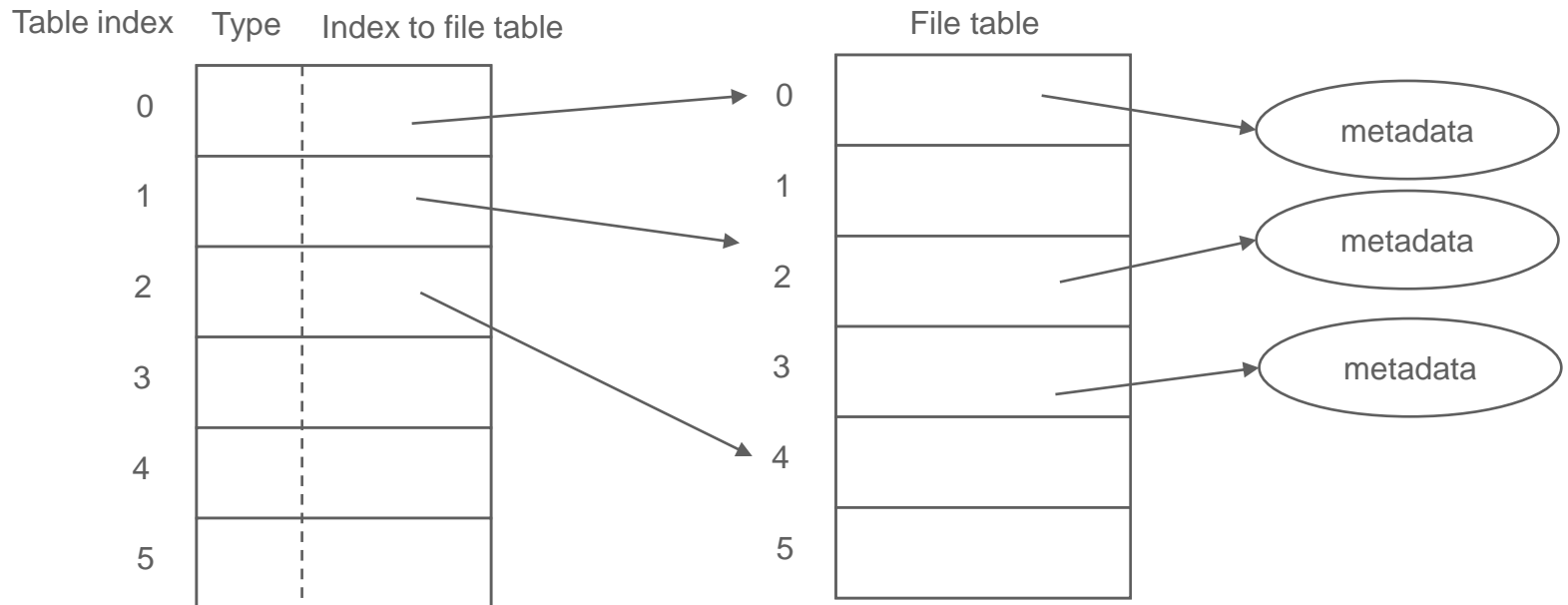
> flag가 `OPEN_FLAG_CREATE`이며, 파일이 존재하지 않으면 생성

File descriptor table

- > open file의 descriptor를 관리하는 table
- > file 객체를 포인트함

File table

- > 해당 파일의 metadata (e.g., inode 번호)를 저장한다.
- > FAT는 inode가 없다. 대신 파일 정보는 directory entry에 저장된다



File Descriptor Table and File Object

```
typedef struct _DescEntry {
    BOOL    bUsed;           // 해당 entry가 사용 중인지를 표시함
    File*   pFile;           // open file의 File object pointer
} DescEntry;

typedef struct __FileDescTable {
    int      numUsedDescEntry; // 사용 중인 entry의 개수
    DescEntry pEntry[MAX_DESC_ENTRY]; // MAX_DESC_ENTRY의 Descriptor entry
} FileDescTable
```

File Descriptor Table and File Object

```
typedef struct __OpenFlag {
    OPEN_FLAG_READONLY,
    OPEN_FLAG_WRITEONLY,
    OPEN_FLAG_READWRITE
} OpenFlag;

typedef struct _File {
    BOOL      bUsed;
    OpenFlag  flag;           // open mode
    int       dirBlkNum;      // 해당 파일 정보가 저장된 디렉토리 블록의 번호
    int       entryIndex;    // 해당 디렉토리 블록에서 몇 번째 entry를 나타내는 index
    int       fileOffset;    // 파일 내에서 최근까지 read/write한 위치(position)
} File;

typedef struct __FileTable {
    int  numUsedFile;
    File pFile[MAX_FILE_NUM];
}
```

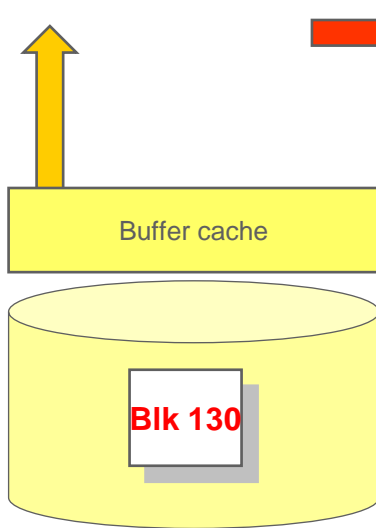
파일 생성

```
int OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
```

> "tmp" 디렉토리 블록을 찾자.

(1) Root 디렉토리의 시작 블록을 확인한다.
1st FAT entry가 130번 블록이다.

(2) 블록 크기의 메모리를 할당하고,
그 메모리로 BufRead 함수를 통해
block 130을 읽는다



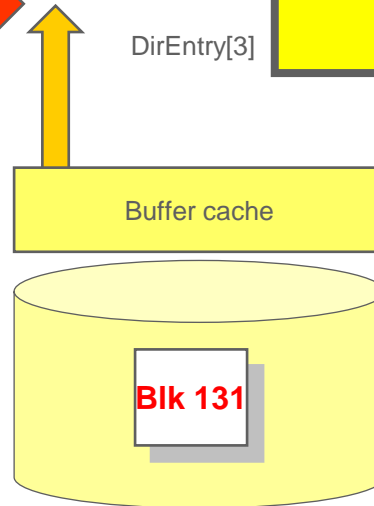
(3) 디렉토리 블록에서 tmp의 디렉토리 블록이
block 131임을 확인했다.

DirEntry[0]	"."	130	...
DirEntry[1]	"tmp"	131	...
DirEntry[2]			
DirEntry[3]			

(5) tmp 디렉토리 블록에 "a.c"가 있는지 검색한다.
없다면, a.c의 directory entry를 추가한다.

DirEntry[0]	"."	131	...
DirEntry[1]	".."	130	...
DirEntry[2]	가!		
DirEntry[3]			

(4) 블록 크기의 메모리를 할당하고,
그 메모리로 BufRead 함수를 통해
block 131을 읽는다



파일 생성

```
int OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
```

> "tmp" 디렉토리 블록에 "a.c"의 directory entry를 추가하자

DirEntry[0]	“.”	131	...
DirEntry[1]	“..”	130	...
DirEntry[2]	“a.c”	-1	...
DirEntry[3]			

(6) DirEntry[2]의 변수들을 설정함.

> name: “a.c”

> Mode: 임의의 값

> startFatEntry: **-1**

> filetype: FILE_TYPE_FILE

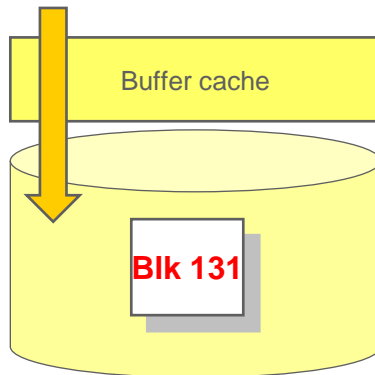
> numBlocks: **0**

파일 생성 시에 파일 블록을 할당하지 않는다.

그래서, startFatEntry는 -1, numBlocks는 0이다.

FileSystemInfo

(7) 변경된 블록을 BufWrite를
사용하여 디스크의 131번 블록으로 저장한다.
Block 131이 tmp의 directory block이다.

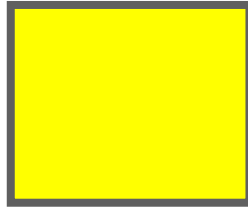


단계 (6)에서 a.c의 directory entry가 포함된
디렉토리 블록 번호 131과 entry 번호 2를 기억해두자!!
다음 단계에서 사용될 것이다

파일 생성

```
int OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
```

> File descriptor table과 file object를 설정하자



(7) Malloc으로 File 객체를 할당한다.



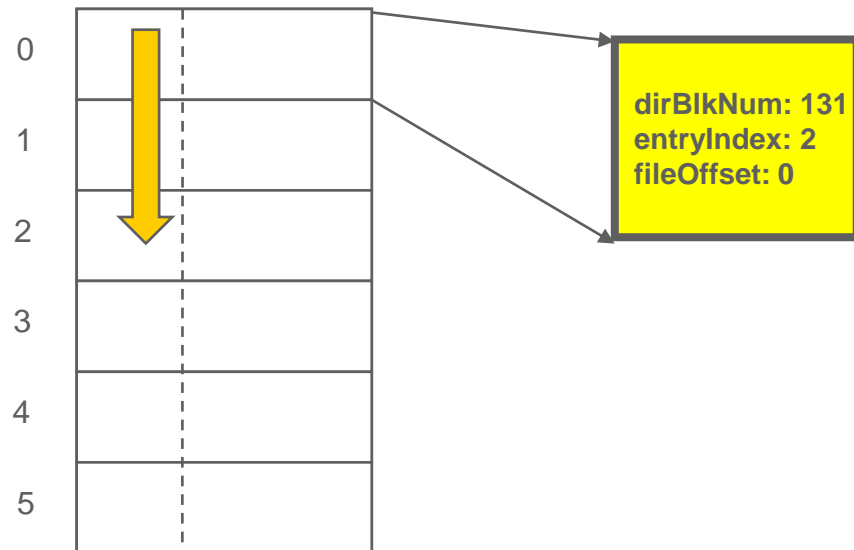
(8) File 객체의 변수들을 설정한다.
단계 6에서 획득한 a.c를 포함한
블록 번호와 entry의 index 번호

File table

0	
1	
2	
3	
4	
5	

0	
1	
2	
3	
4	
5	

(9) File descriptor table의 index 0부터 시작하여,
빈 entry를 찾는다. 찾았다. Index 0이라 가정하자



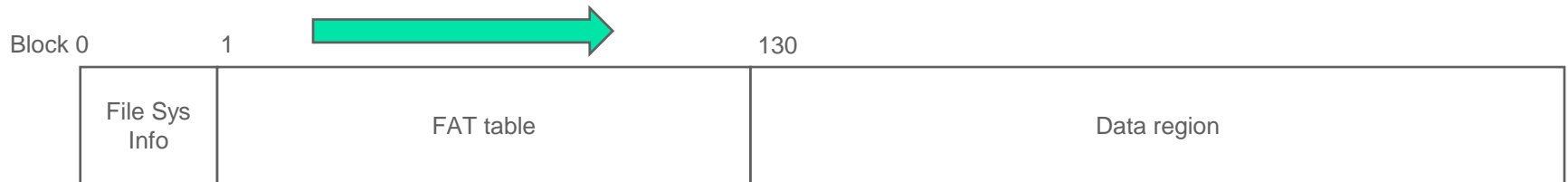
(10) File table에서 빈 엔트리를 검색한다. Entry 0을 찾았다.
이때, File descriptor table의 index 0에 file table entry 0을 저장한다.

파일 쓰기

int WriteFile(fd, pBuf, BLOCK_SIZE):

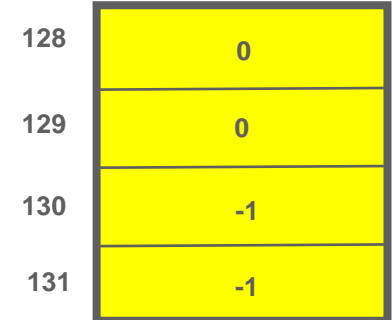
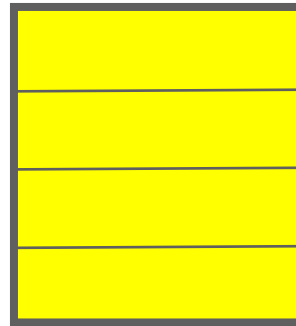
> FAT table에서 빈 'entry'를 할당함. 이 entry의 index는 데이터를 저장할 블록임

(1) FAT table의 130번 entry부터 빈 entry를 검색함. 검색된 entry 번호가 **132**이라고 가정함.

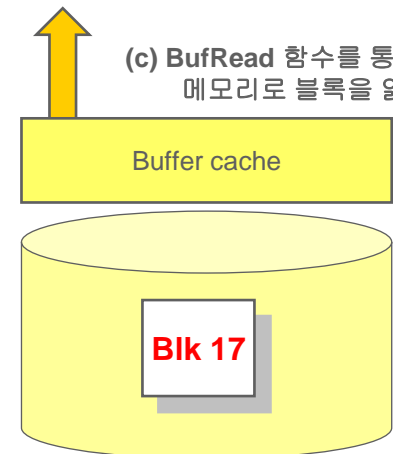


(a) Entry 130이 포함된 FAT 블록을 확인한다.
130은 16번째 FAT 시작 블록에서 16번째 블록이다.
블록 번호: $1 + 16 = 17$ 번

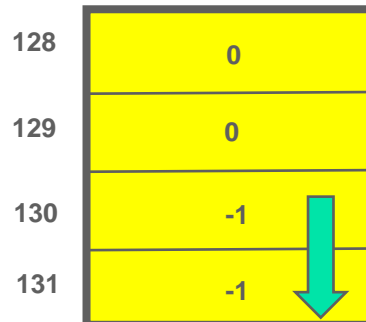
(b) Block 크기의 메모리를 할당한다.



(c) BufRead 함수를 통해 할당된 메모리로 블록을 읽는다



(d) 130번부터 130까지 빈 entry를 검색했지만,
발견할 수 없었다. 그러면 132를 검색하자.



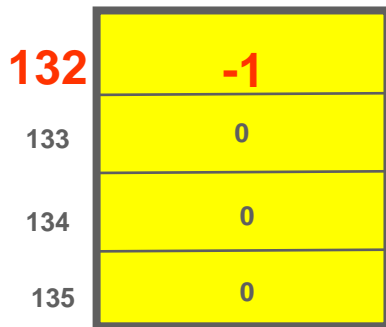
파일 쓰기

int WriteFile(fd, pBuf, BLOCK_SIZE):

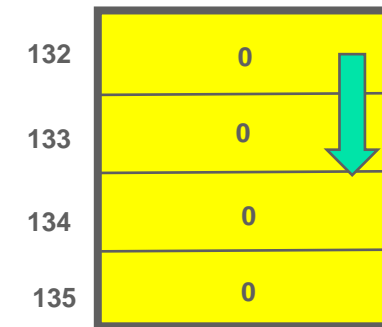
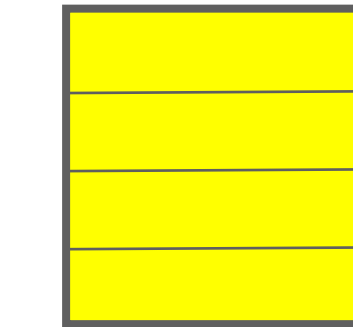
> FAT table에서 빈 entry를 할당함. 이 entry의 index는 데이터를 저장할 블록임

(e) 132번이 저장된 블록은 block 18임을 확인한다.

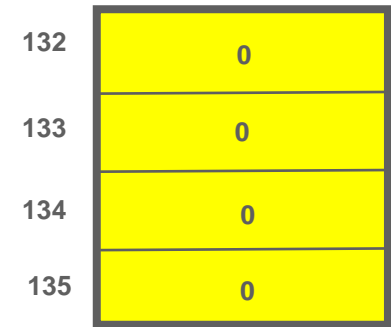
(f) Block 크기의 메모리를 할당한다.



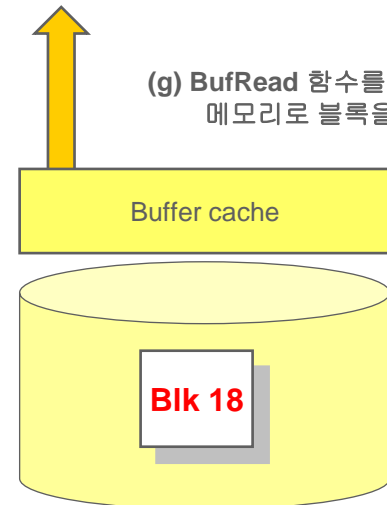
(e) 파일 블록을 할당해야기 때문에 Entry를 -1로 변경하고, BufWrite 함수를 통해 디스크로 block 18을 저장함



(h) 132번부터 빈 entry를 검색한다.



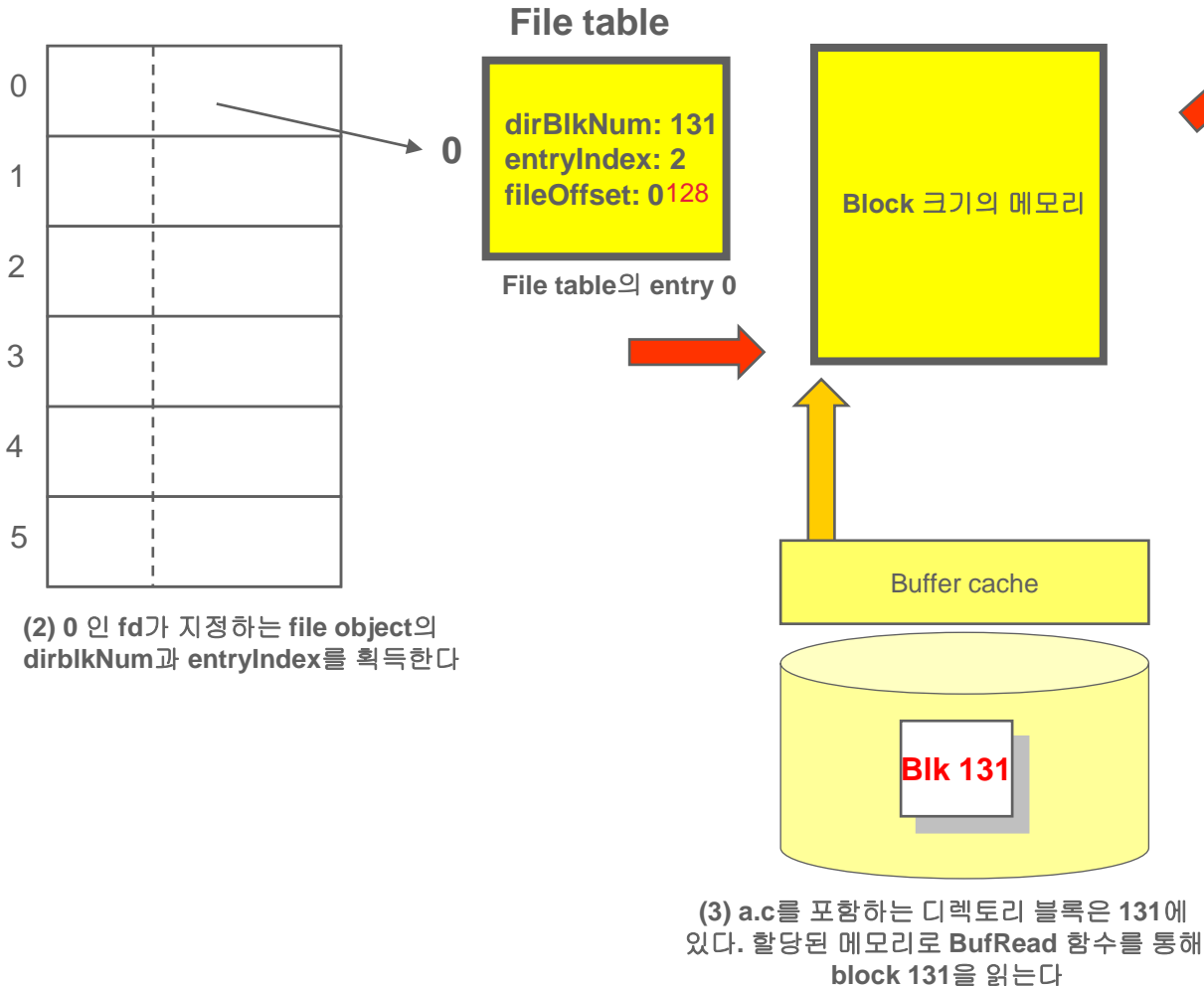
(g) BufRead 함수를 통해 할당된 메모리로 블록을 읽는다



파일 쓰기

int WriteFile(fd, pBuf, BLOCK_SIZE):

> a.c의 directory entry에 있는 변수들을 설정하자



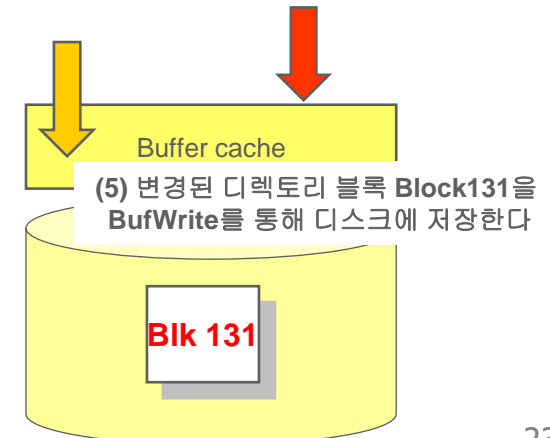
DirEntry[0]	"."	131	...
DirEntry[1]	..	130	...
DirEntry[2]	"a.c"	-1	...
DirEntry[3]			

DirEntry[0]	"."	131	...
DirEntry[1]	..	130	...
DirEntry[2]	"a.c"	132	...
DirEntry[3]			

(4) DirEntry[2]의 변수들을 설정함.

> startFatEntry: 132, numBlocks: 1

entry 2는 entryIndex로 확인



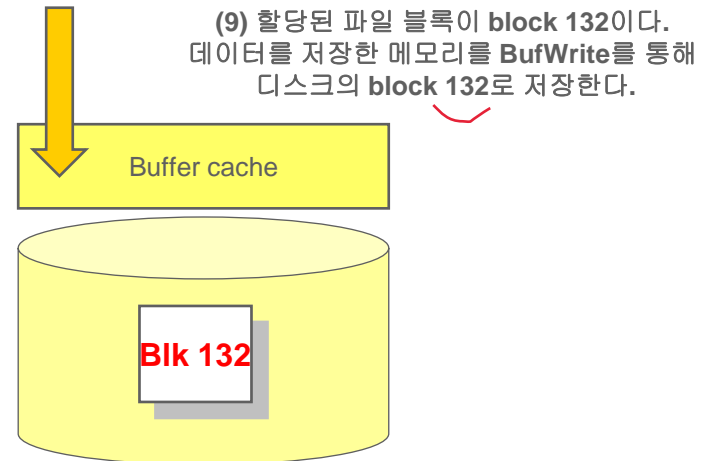
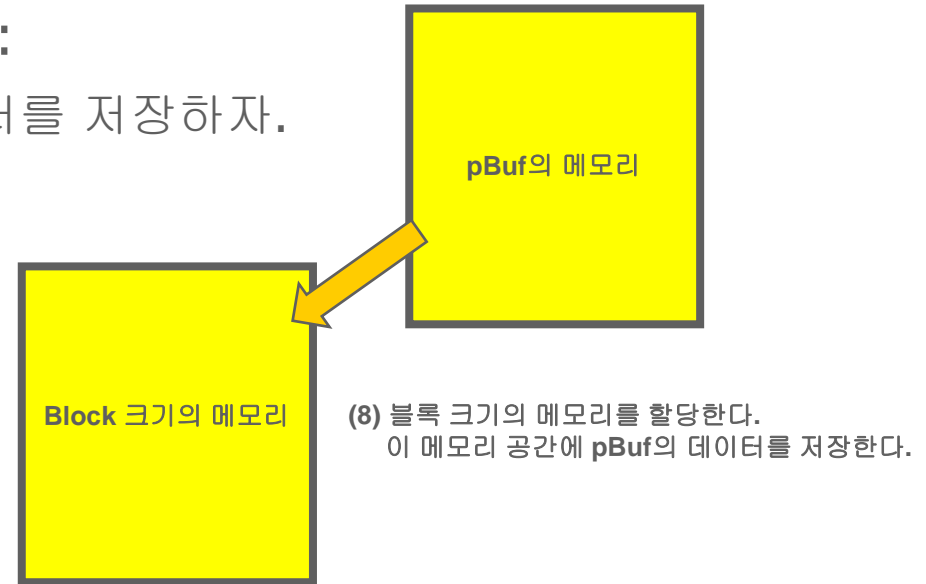
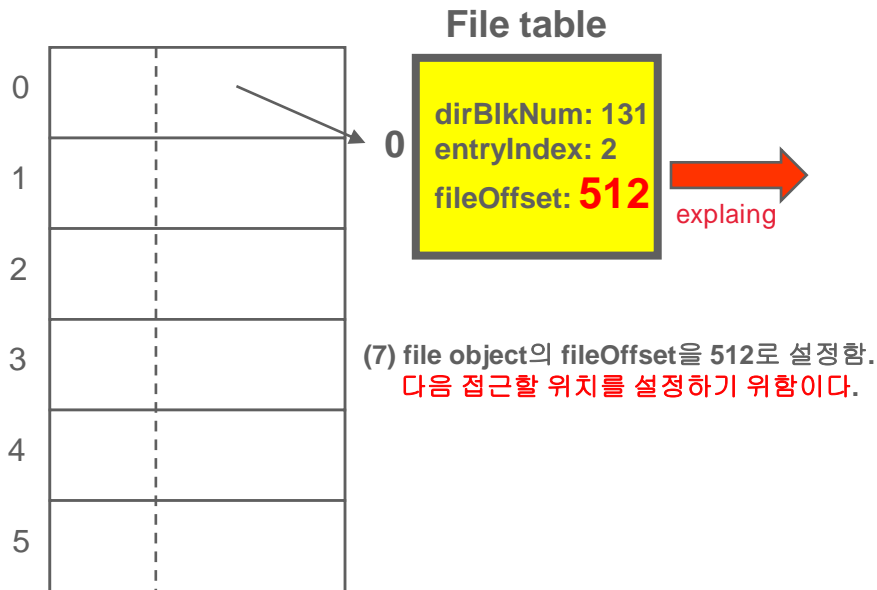
파일 쓰기

int WriteFile(fd, pBuf, BLOCK_SIZE):

> FileSysInfo를 업데이트하고, 데이터를 저장하자.

(6) Block이 할당되었기 때문에 FileSysInfo를 업데이트함

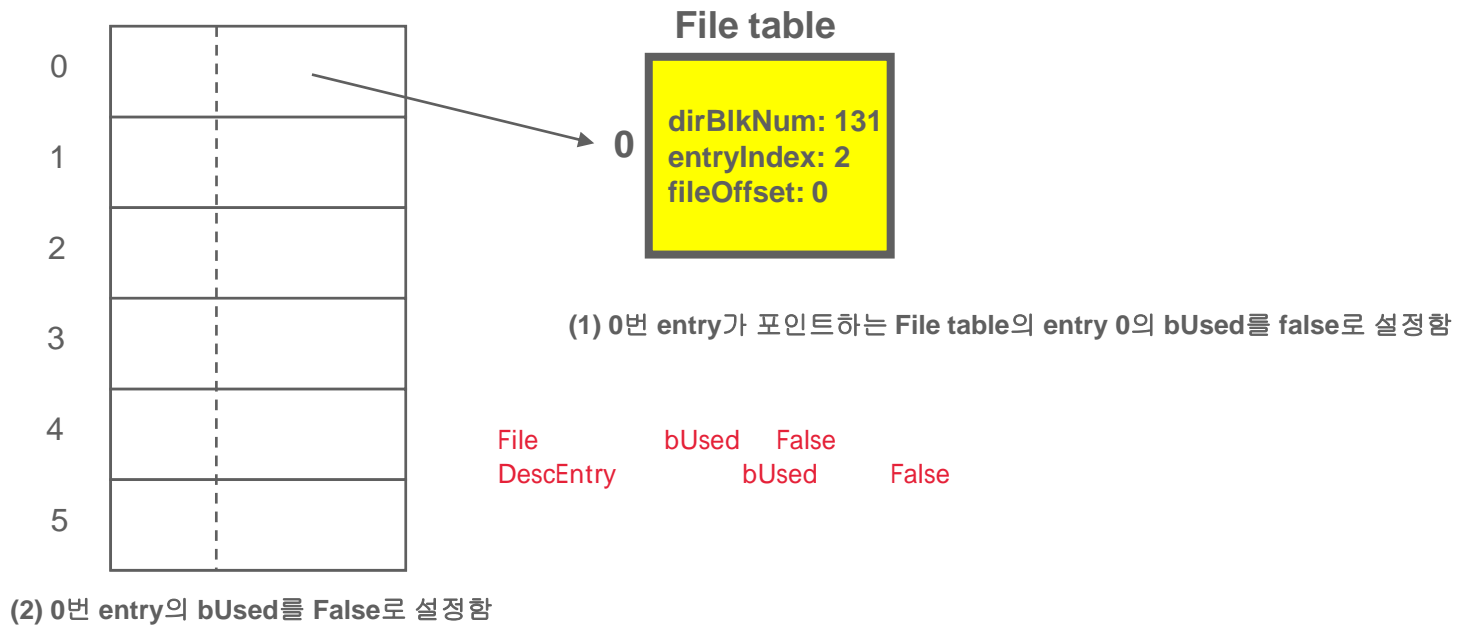
> 방법은 14쪽 참고함



파일 닫기

CloseFile(0)

> 0번 descriptor를 닫고자 한다면, file object를 free하고, 0번 entry의 bUsed를 False로 설정

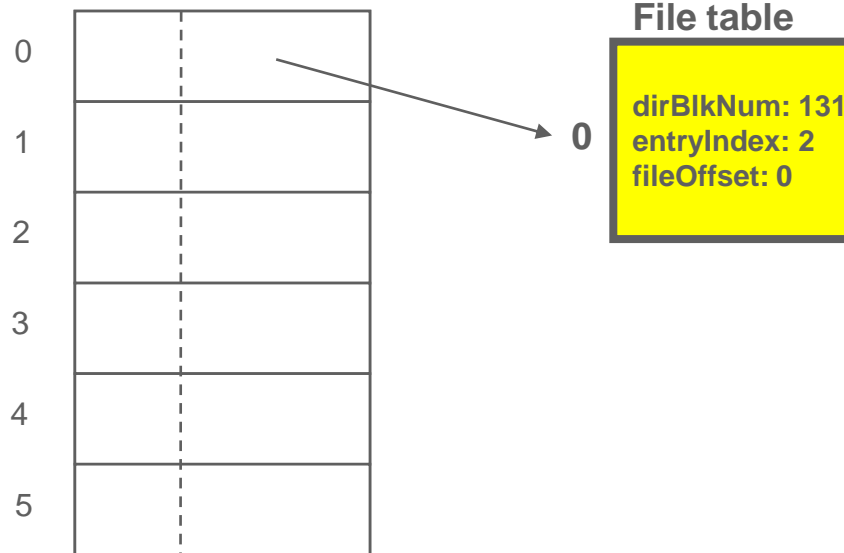


파일 열기

```
int OpenFile("/tmp/a.c", OPEN_FLAG_CREATE)
```

> a.c가 이미 존재하기 때문에 tmp에서 a.c를 포함하는 블록번호와 directory entry의 번호를 읽는다. 아래 그림처럼 설정해주면 된다.

Tmp > 17쪽, 단계 5를 참고하자.
a.c



(9) Entry 0의 bUsed를 True로 설정하고,
file object를 포인트한다.

파일 삭제

RemoveFile("/tmp/a.c"):

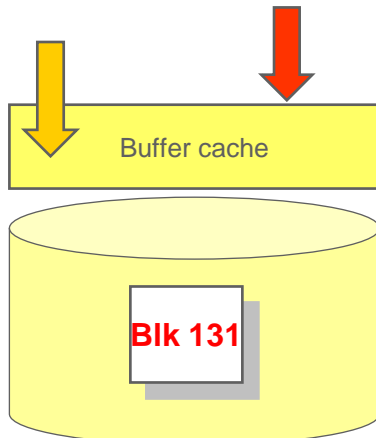
- > 이미 존재하는 파일을 열기하는 동작처럼 a.c의 directory entry를 찾는다.
- > directory entry의 startBlockNum을 인자로 FatRemove 함수를 호출한다.
FatRemove(firstblock, startblock) firstblock, startblock에 어떤 값을?
- > tmp 디렉토리 블록에서 a.c의 directory entry를 삭제한다.
- > 파일이 한 개 삭제되고, 파일의 블록들이 모두 삭제되었다. FileSysInfo를 변경한다.

(1) 디스크에서 a.c를 포함하는 tmp의 디렉토리 블록을 읽는다

DirEntry[0]	"."	131	...
DirEntry[1]	".."	130	...
DirEntry[2]	"a.c"	132	...
DirEntry[3]			

(2) FatRemove 함수를 사용하여 할당된 entry들을 해제한다.

132	-1
133	0
134	0
135	0



FatRemove 함수를 호출하기 전후에 디스크를 접근하는 동작을 할 필요가 있는가?
> 없다. 왜냐하면, FatRemove 함수는 BufRead, BufWrite를 호출하여 FAT table의 블록들을 읽고, 변경된 블록들을 저장하기 때문이다.

(3) 변경된 디렉토리 블록 Block131을 BufWrite를 통해 디스크에 저장한다

(4) FileSysInfo를 적절히 변경하고
디스크 Block 0으로 저장한다.

디렉토리 삭제

RemoveDirectory("/tmp")

- > "/" 밑에 디렉토리 엔트리를 삭제하는 동작은 파일 삭제하는 동작과 동일함.
- > 디렉토리를 구성하는 디렉토리 블록들을 **FatRemove** 함수를 호출하여 삭제함
즉, **directory entry**의 **startBlockNum**을 인자로 **FatRemove** 함수를 호출한다.
- > 단, 빈 디렉토리만 삭제 가능함. 디렉토리에 파일이 있으면 삭제 불가능.

디렉토리 열기, 읽기 및 닫기(Linux)

```
Struct dirent* dentry; /* dirent structure */
DIR* dirp; /* directory pointer (descriptor) */

if ((dirp = opendir("/home/park")) == NULL) {
    perror("opendir");
    exit(1);
}
while ((dentry = readdir(dirp)) != NULL)
{
    printf("name:%s, type:%d\n",
          dentry->name, dentry->type);
}
if (closedir(dirp) < 0) {
    perror("closedir");
    exit(2);
}
```

디렉토리 열기와 읽기

Directory* OpenDirectory(char* name)

- > 디렉토리를 open하는 함수. 리눅스의 opendir() 동일함
- > 성공하면 Directory의 주소를 반환, 실패하면 NULL 반환
- > 구현 방법: name의 디렉토리를 검색하고, Directory 구조체를 동적으로 메모리로 할당. 할당된 Directory에 dirBlkNum, entryIndex를 저장한 후 메모리 주소를 반환함.

```
typedef struct __Directory {  
    int    dirBlkNum;    // 해당 디렉토리 정보가 저장된 디렉토리 블록의 번호  
    int    entryIndex;  // 해당 디렉토리 블록에서 몇 번째 entry를 나타내는 index  
} Directory;
```

```
tmp      : 130  
          : 131  
dirBlkNum 130  
entryIndex 1
```

0	.	130
1	tmp	131
2		
3		

디렉토리 열기와 읽기

`FileInfo* ReadDirectory(Directory* pDir)`

- > 디렉토리에 저장된 파일의 정보를 획득함. 리눅스의 `readdir()` 동일함
- > 디렉토리에 포함된 **File**의 정보를 순차적으로 획득함. 성공하면 **FileInfo**의 주소를 반환. 더 이상 획득한 파일 정보가 없으면, **NULL** 반환
- > 구현 방법: **pDir**에 저장된 **dirBlkNum**, **entryIndex**를 사용하여 디렉토리 블록을 검색. 해당 디렉토리 블록에 저장된 각 파일의 정보를 **FileInfo**의 구조체에 저장하여 반환함. 이때, **FileInfo** 구조체를 동적 메모리로 할당하고, 이 주소를 반환함.

```
typedef struct _FileInfo { // 리눅스에서 dirent와 유사함
    char*      name[16];    // 파일 이름
    AccessMode mode;        // file access mode, read-only, write-only, read-write
    int startFatEntry;      // file의 1st fat entry 번호
    FileType filetype;      // file type
    int numBlocks;          // file을 구성하는 블록 개수
} FileInfo;
```

Directory index position ...?

Blk 131

DirEntry[0]	"."	131	...
DirEntry[1]	".."	130	...
DirEntry[2]	"a.c"	132	...
DirEntry[3]			

디렉토리 닫기(Linux)

int CloseDirectory(Directory* pDir)

- > 열린 디렉토리를 닫는 함수. 리눅스의 `closedir()` 동일함
- > 구현 방법: `pDir`가 가리키는 메모리를 해제함. 성공하면 0, 실패하면 -1을 반환함. 단, 실패하는 경우는 테스트케이스에 포함하지 않을 계획.

```
FileInfo* pFileInfo;
Directory* pDir;

If ((pDir = OpenDirectory("/home/park")) == NULL) {
    perror("OpenDirectory");
    exit(1);
}
While ((pFileInfo = ReadDirectory(pDir)) != NULL)
{
    printf("name:%s, type:%d\n",
        pFileName->name, pFileInfo->fileType);
}
If (CloseDirectory(dirp) < 0) {
    perror("CloseDirectory");
    exit(2);
}
```