

- file System

- goal
- example fs
- EXT2 fs

- file system goal : store file in disk
***efficiently**

→ 목표 : 1. Time efficiency

특정 파일의 모든 block number
찾아내는 속도!!!
(block이 정렬하는 속도와
따라.)

file name
↓ ... how many
location
(block location)

block을 가리기
하기 위해 disk
access 한다.

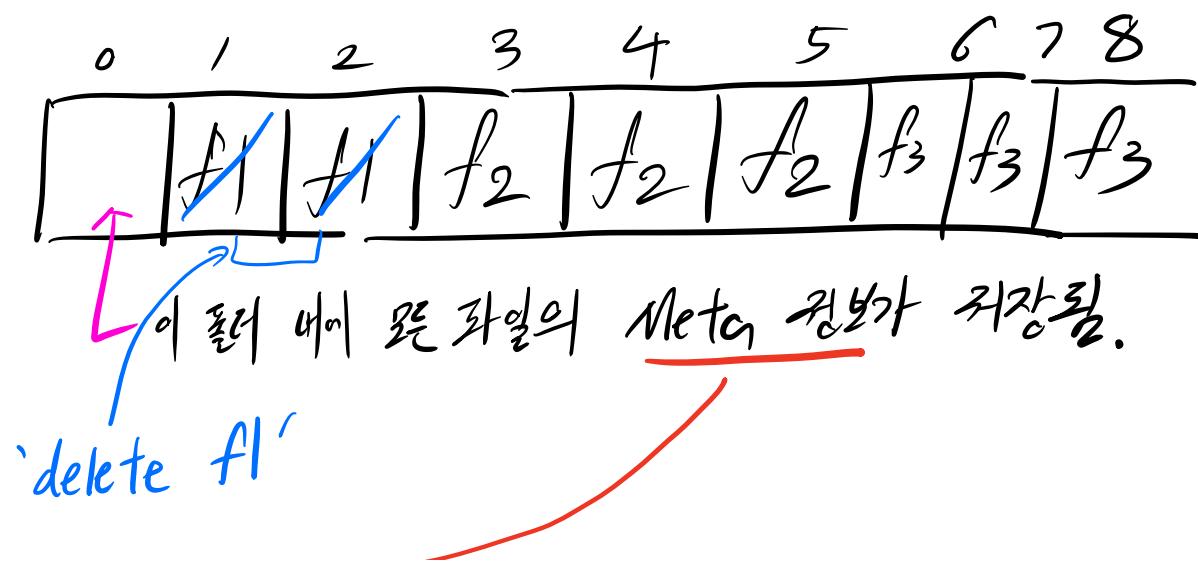
2. Space efficiency : 쓸 데 없는 공간을
활용하지 않는 것.

⇒ ②. 많은 양의 데이터를 끌어와 처리한,

① 이렇게 되면 꺼내는 시간이 많이 소모된다.'

- Example fs
 $\text{file} = \sum \text{block}$
 $\text{disk} = \sum \text{block}$
 $1 \text{ block} = 1 \text{ Kbyte}$
 write $f_1(2 \text{ block})$
 write $f_2(3 \text{ block})$
 delete f_1
 write $f_3(3 \text{ block})$
- (= 디스크 조작')
- '블락'들로 나누어진 디스크에 파일을 저장하기 위해,
 파일도 '블락' 단위로 쪼갠다!
- 디스크 공간을 '블락' 크기로 쪼갠다.
 ↗️ 이제 핵심이라!!
- just assumption for
 following example

① Consecutive allocation fs



b ϕ (block for directory file)

f_n	block start	Size
f_1	1	2
f_2	3	3
f_3	6	3

↑ 이를 통해, '디렉토리(폴더)'
도 하나의 파일로 취급되는
것을 알 수 있다.
일반적인 파일 block에는
데이터들이 저장되어 있다면,
'디렉토리 파일'의 block에는
디렉토리에 들어 있는 파일의
파일정보가 저장되어 있다.

f_3 의 효율성은?
(f_3 block은 6, 7, 8 byte!)

T : 1 (=good!)

block φ'이란
저장하면 됨.

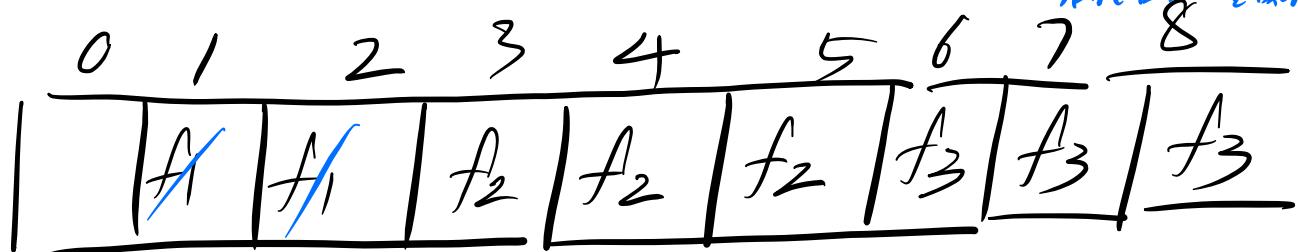
S : 낭비가 존재함!

→ f_1 이 존재했던

block이 f_3 이
못들어갈!

size : 3 block

② linked list allocation fs ⇒ 각 block에는 다음 block의 위치(주소)로 들어온다.



$b \phi(\text{dir})$

⇒ 각 block의 첫번째 위치만 알 수 있다.

f_n	block start	size
f1	1	2
f_2	3	3
f_3	1	3

⇒ block 01
접근하여, 2
다음 block의
link를 물어
보자면
재운다!

f_3 의 효율성은?
T: 3번!
(-비효율적!)

$S: 0$ (good!)

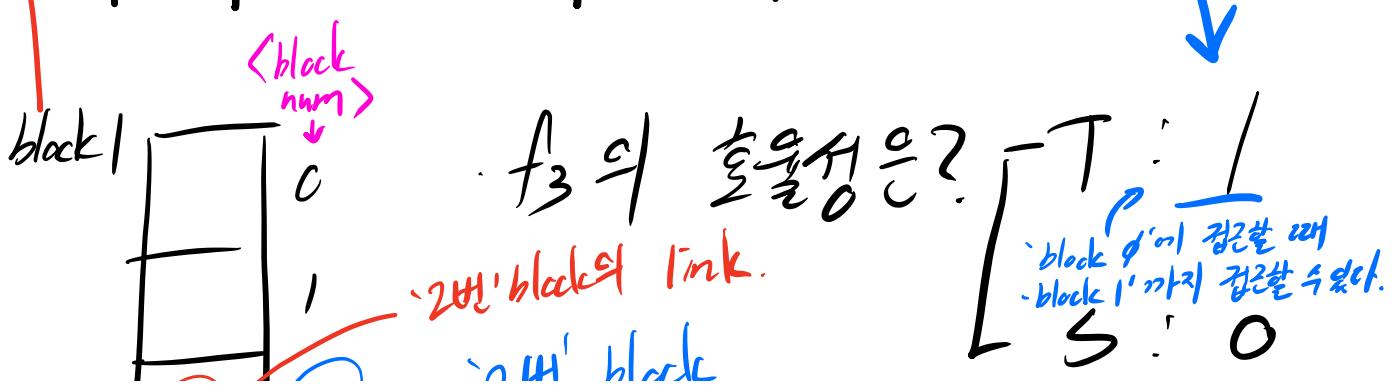
③ FAT (File Allocation Table)

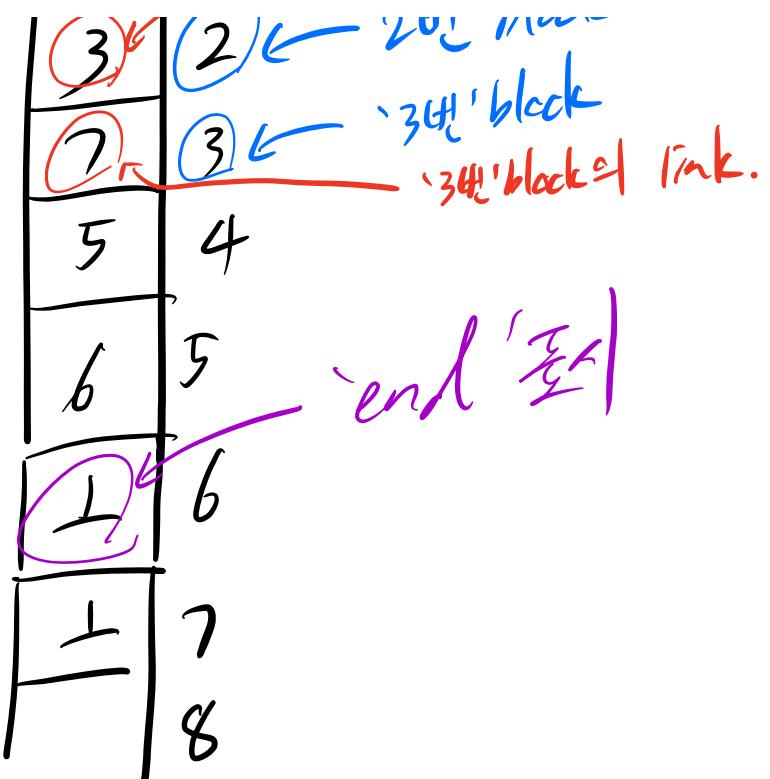
0	1	2	3	4	5	6	7	8	9
	f1	f1		f_2	f_2	f_2	f_3		
	f_3	f_3							

bq (dir) \overline{T} (=link table)

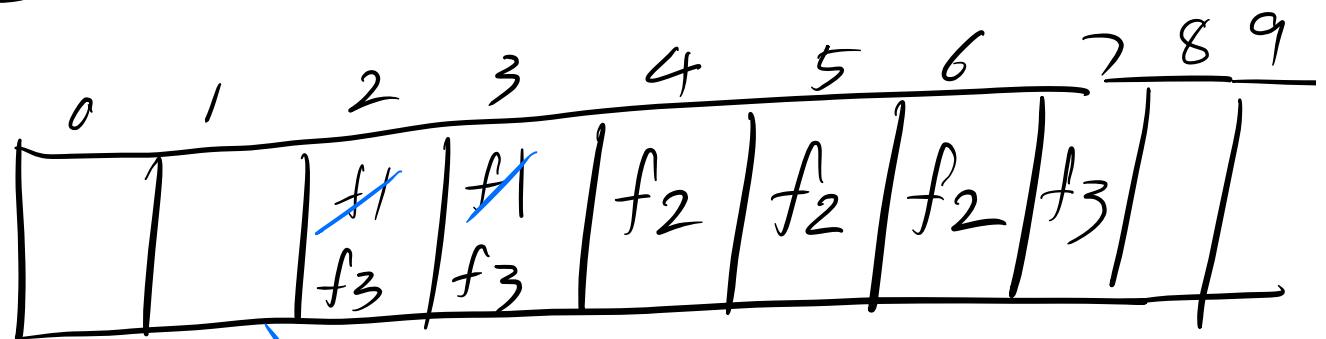
fn	block start	size
f_1	2	2
f_2	4	3
f_3	2	3

핵심!!!!
 디스크 내 블락이 접근하는 한 번의
 시도에서, 인접한 블락까지 접근할
 수 있다.





④ inode file system.



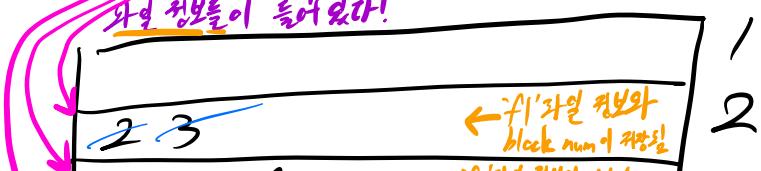
$b\phi(\text{dir})$

b1 (inode table)

inode 번호.

↑ 각 'inode array'에는 block 번호와 파일 정보들이 들어 있다!

fn	inode 번호
----	----------

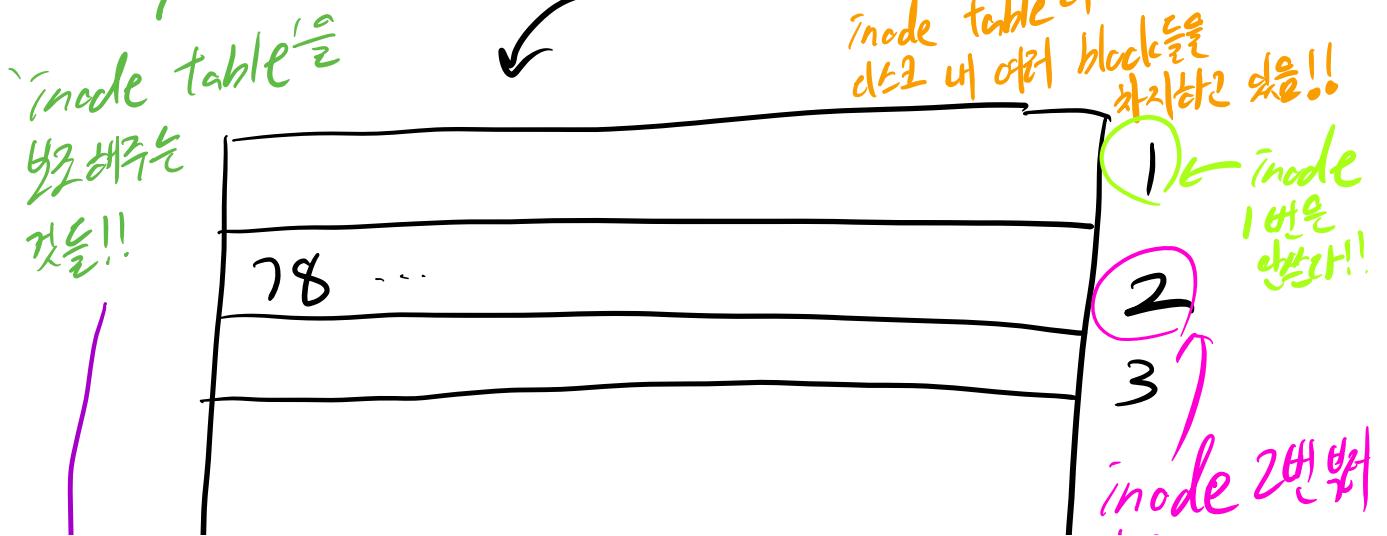
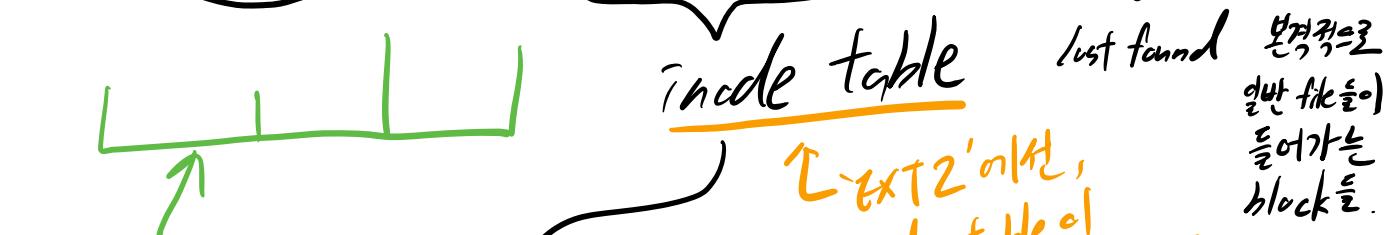


<i>f1</i>	2	4 5 6 2 3 7	3 4
<i>f2</i>	3		
<i>f3</i>	4		

한 개 inode 크기 : 128 byte

⑤ EXT2 fs (Linux 전용 fs) (assumption, 1 block size = 1024)

< 1 block size = 4K byte >



사용!

S → `EXT 2'에 대한 기본적인 정보
(ex) total number of Inodes, total number of blocks, block size...)

G → IBM, DBM, inode table의 주소상 위치 정보
D → DBM (Data block Bitmap):

어느 블럭이 사용되고 있는지에 대한 정보

I → IBM (Inode Bitmap):

어느 inode가 사용되고 있는지에 대한 정보

X block의 위치(주소)를 나타내는 단위: h.

typedef struct {

총 inode의 개수

u32 m_inodes_count; // 0-3

u32 m_blocks_count; // 4-7

↑ 총 block의 개수

u32 m_first_data_block; // 14-17

// block location
of Superblock

u32 m-log-block-size; // 18-1B

↑ 한 개의 block size

u16 m-magic; //

...

} Superblock.

X DBM과 IBM의 한 바이트 내
기입된 것은 16진수이다. 이를 이진수로
변환해야 한다. '1'의 총 개수가
사용되었는 총 block 또는inode 개수이다.

• inode table

inode

file

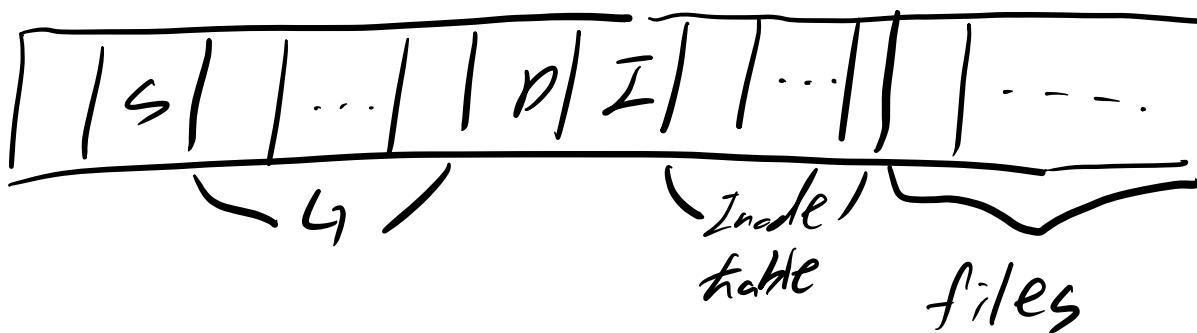
directory file → 파일 목록
다음과 같다!!

root inode

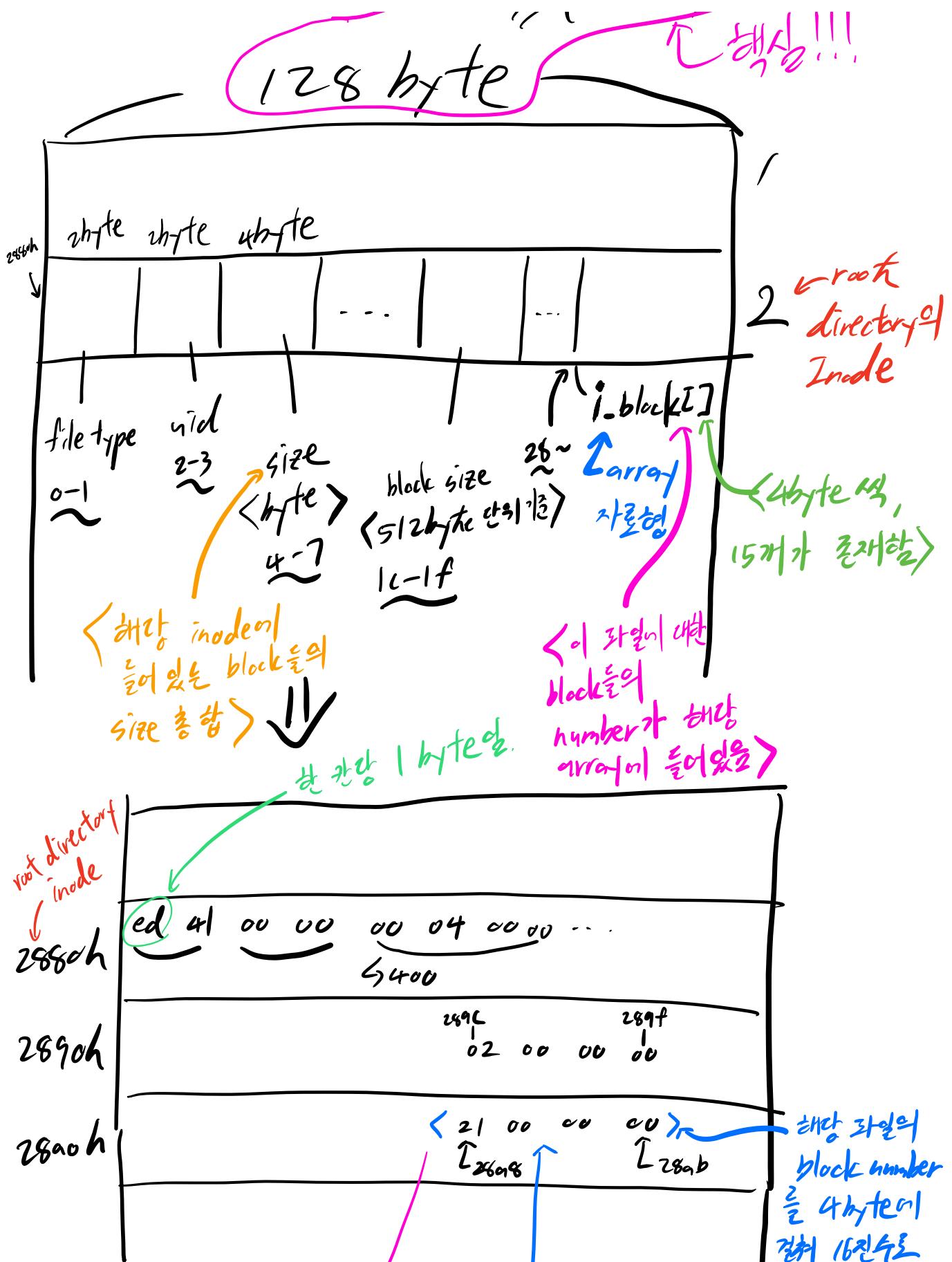
root dir file

file creation

file removed



• Inode table ~ (0x80h)



' 나타내.

$0x21 = 33 \Rightarrow$ 해당 파일의
1진수 10진수 block number는
33!!.

block 33은 '33792byteon'
위치함.

$$33792 = 0x8400$$

* root directory의 block (block 33)

8400 02 00 00 00 00 00 01 02

.file

- data file
 - text file
 - non-text file (binary file)
 - (ex) .exe, .war, .jpg ...)

directory file

(inode number, record length, name length)

ex) root directory file block
(block 33)

directory file block 내 정의

이를 먼저 보면 좋겠어요!!

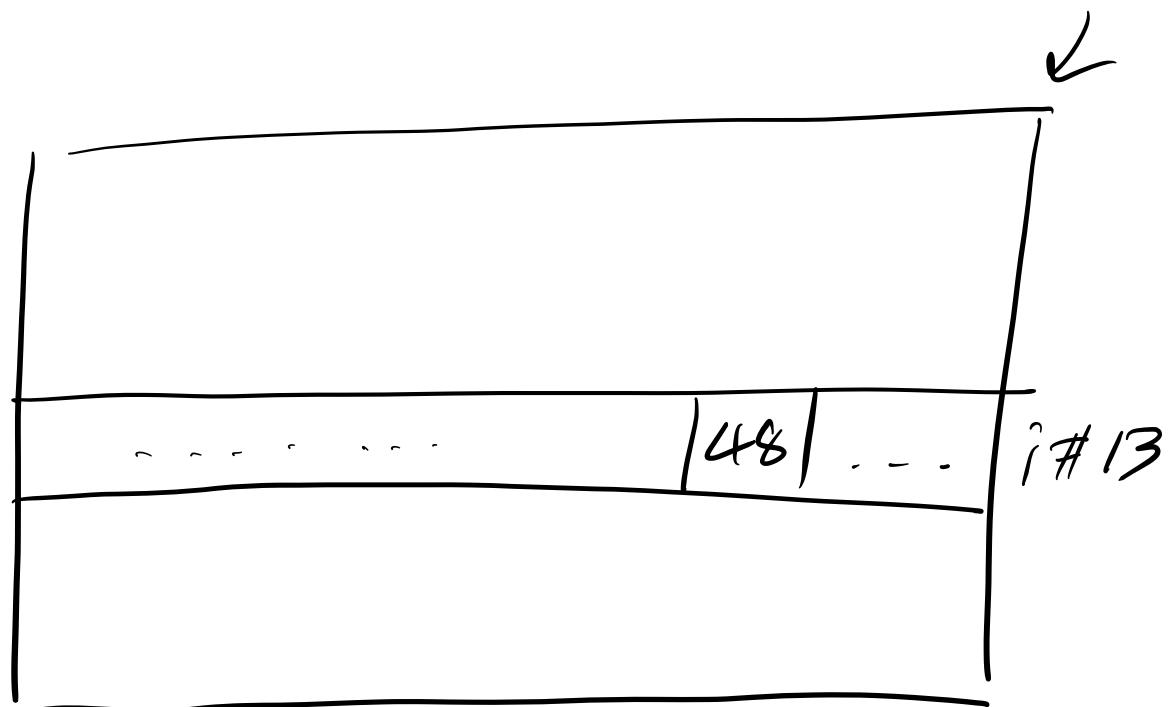
inode num(4byte)	record-length(2byte)	name-length(1byte)	file-type(1byte)	file-name
2	12			<i>자기 자신을 찾는다.</i>
2	12			<i>부모 디렉토리를 찾는다.</i>
11	20			lost found
12	<i>아~~~ 끝까지!!</i>			f1

• Make another file : f2

① allocate inode number : inode num 13
⇒ IBM

② " a block : block 48
⇒ DBM

③ allocate inode : inode table



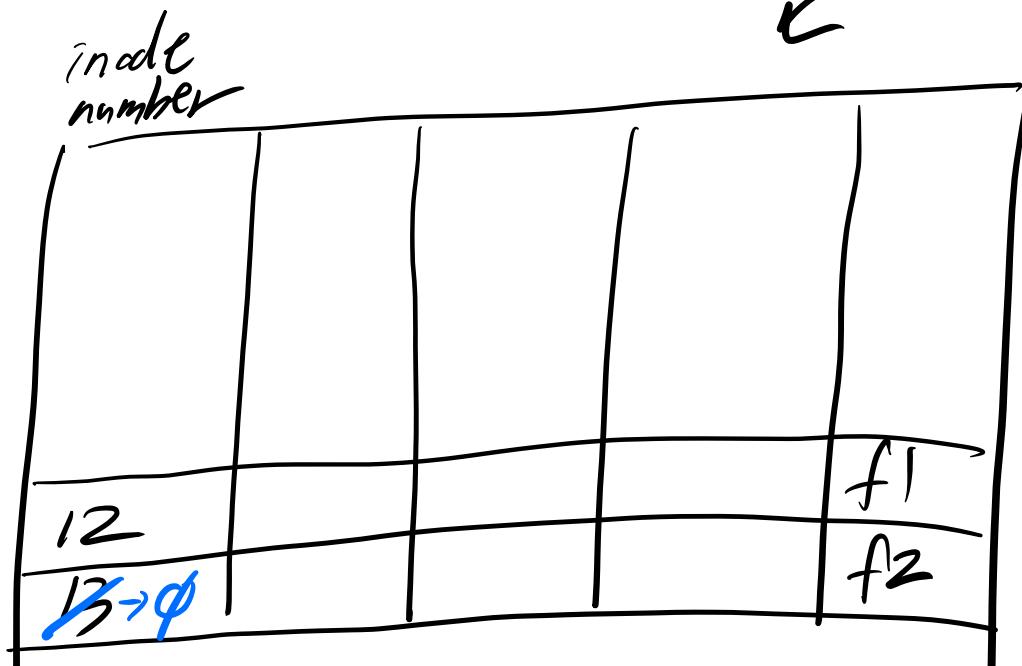
④ Insert in parent dir file

⑤ block 48

hello.

- remove f_2

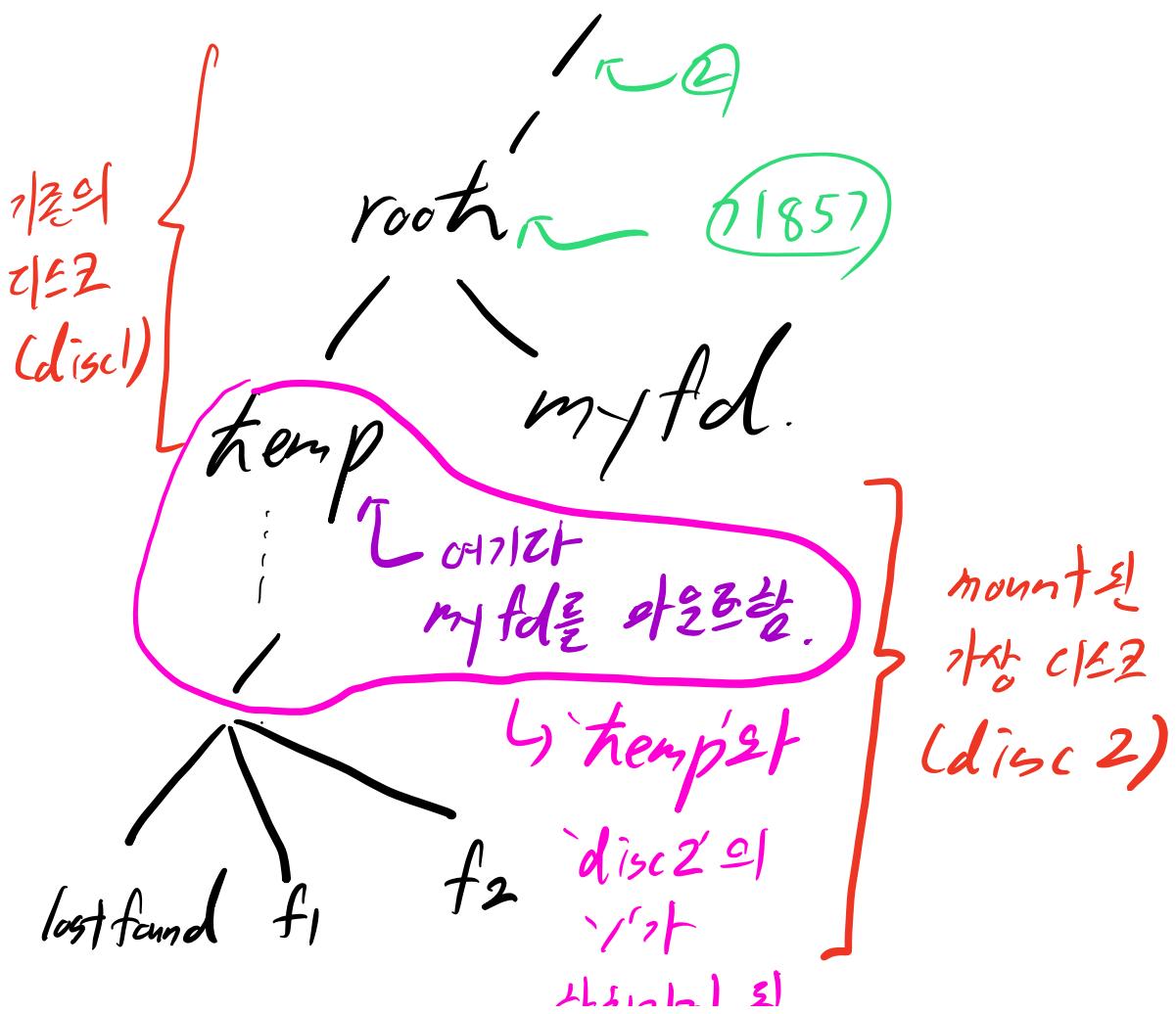
- ① remove inode 13 in IBM
- ② remove block 48 in PBM
- ③ remove f_2 in root directory



l

J

X 파일을 삭제하면, 해당 파일의
inode number는 0으로 바뀐다.
나머지 것들은 224로 남아있다!!



교수님이 쓴 것.

- mount된 후의磁盤 tree 형태.

