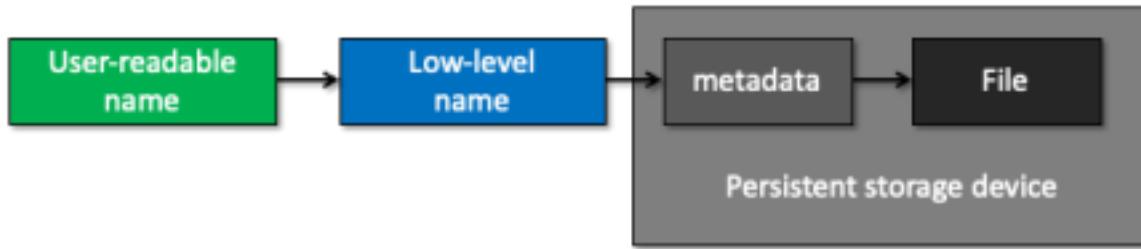


* open, close, read, write, lseek ⇒ 사용자 반복의 결과
* fcontrol

Chapter 29. Interlude : Files and Directories

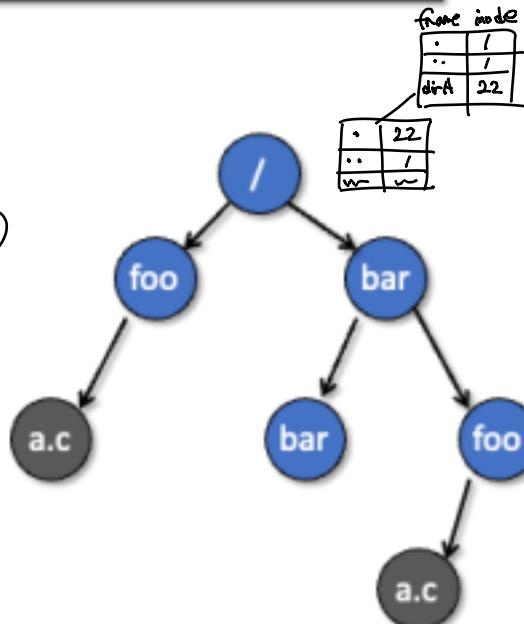
(1) Files

- 바이트의 선형 배열
- 사용자는 선형 배열을 읽고 쓴 (시스템에서 보는 파일)
- 사용자가 읽을 수 있는 이름 (fname) vs 저수준 이름 (inode 구조체)
- 인간이 읽을 수 있는 이름 (string)
- 저수준 이름 : inode (index node). → vnode (virtual inode).
- ✓ 다양한 파일 시스템을 지원
 - UNIX : UFS, System III. → S IV. → System V. S5FS, RFS (Remote) / NFS (Network File System)
 - BSD -> FastFS (FFS) ⇒ 초기 속도가 매우 빠름
 - Linux - ext2 -> ext4, XFS (Journaling FS)...
 - DOS File system - FAT16 (file allocation table) -> FAT32
 - Windows - NTFS
 - Mac - HFS -> HFS + -> APFS
 - Coda FS, AFS
 - Proc FS : 특수 파일 시스템. 메모리 내 존재
- 영구히 저장할 수 있는 디바이스에 저장된 콘텐트의 주소를 포함



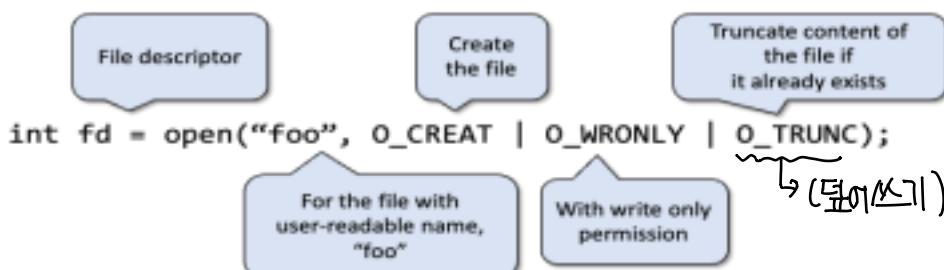
(2) Directories

- 파일과 같이 저수준 이름이 있음 (inode). → inode structure
- (사용자가 읽을 수 있는 이름, 저수준 이름) 쌍들의 리스트 포함
- 각 쌍들은 디렉토리의 한 항목을 표현 (filename, inode 번호)
- 일반적으로 디렉토리는 계층적 구조



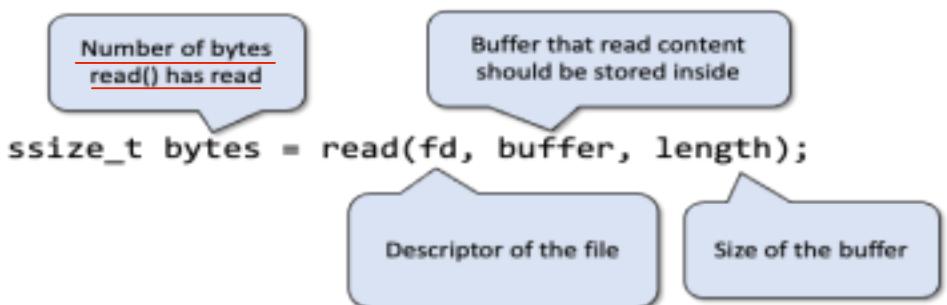
(3) Creating Files

- * open() system call : 파일 생성 / 파일 오픈
- 파일의 file descriptor 리턴
 - 파일을 처리하기 위한 int (index)



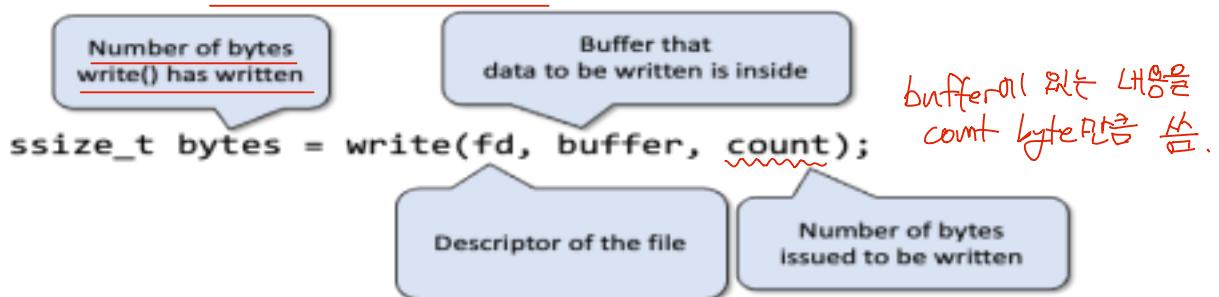
(4) Reading Files

- `read()` system call : 파일을 읽고 읽은 내용을 버퍼에 저장



(5) Writing Files

• write() system call : count 만큼의 바이트를 파일에 씁



(6) File Descriptor and Read/Write File Offset

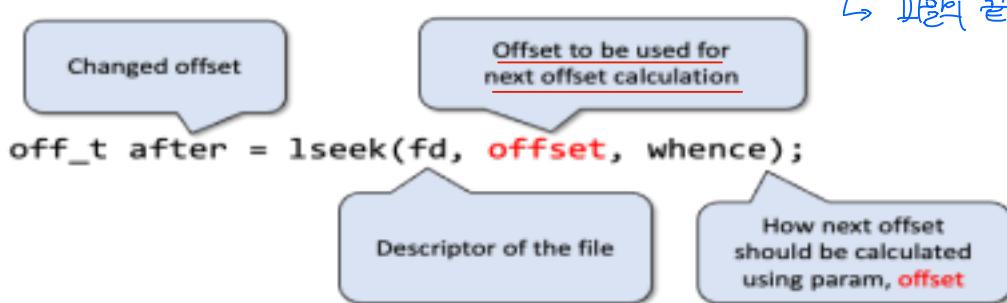
- 파일 시스템은 각 file descriptor 의 read / write offset 유지 (어디까지 읽고, 썼는지)

- read() / write() system call does read from / write to the offset
 - The offset be increased as read() / write() system call do read / write

(7) File Offset Changing

 lseek() system call : 파일의 오프셋 변경 파일을 읽고 쓰는 위치를 지정하는 시스템콜

- If whence is SEEK_SET, the offset is set to offset bytes. → 파일의 시작점을 기준으로 offset값을 이동
 - If whence is SEEK_CUR, the offset is set to its current location plus offset bytes. → 현재 offset 위치에서 offset만큼 이동
 - If whence is SEEK_END, the offset is set to the size of the file plus offset bytes.



⑧ Writing Immediately → $f_{sym}(C)$ 을 호출해서 즉시 디스크에 쓸 수 있다.

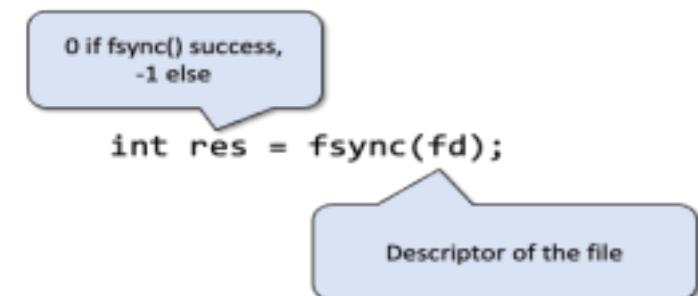
- Write() does writes into buffer, not to the storage device directly ⇒ ~~디스크에~~ 썼다가 → flush ⇒ 디스크에 저장
 - Delayed write (지연쓰기) 버퍼 SW 캐쉬 → UNIX ... 10 초마다. Sync() 버퍼_> 디스크 flush

- 10초 사이에 전원 결함이 생기면.. 파일의 일부를 잃을 수 있다. \$ sync \$ sync \$ halt

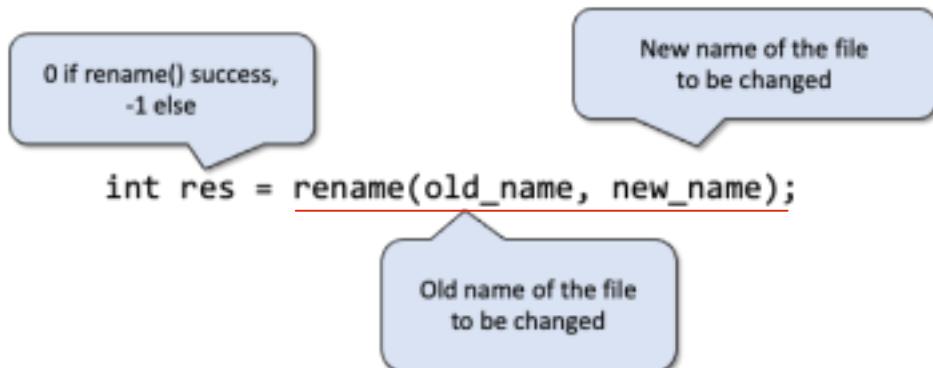
- Writing be done into the device eventually in batching manner

- Data could be lost if the system crash after write() call return and before the batched writing finished

- fsync() system call requests the batched writing be done immediately



(9) Rename Files



(10) Getting Information about Files

* stat() system call : 파일의 정보 리턴

- 대부분의 정보는 inode 구조체로부터 추출

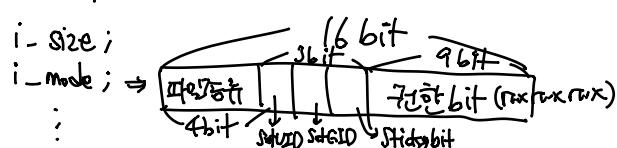
ls 명령 사용 시 이 구조체를 활용한다.

```
struct stat {  
    dev_t      st_dev;      /* ID of device containing file */  
    ino_t      st_ino;      /* inode number */  
    mode_t     st_mode;     /* protection */  
    nlink_t    st_nlink;    /* number of hard links */  
    uid_t      st_uid;      /* user ID of owner */  
    gid_t      st_gid;      /* group ID of owner */  
    dev_t      st_rdev;     /* device ID (if special file) */  
    off_t      st_size;     /* total size, in bytes */  
    blksize_t  st_blksize;  /* blocksize for file system I/O */  
    blkcnt_t   st_blocks;   /* number of 512B blocks allocated */  
    time_t     st_atime;    /* time of last access */  
    time_t     st_mtime;    /* time of last modification */  
    time_t     st_ctime;    /* time of last status change */  
};
```

hard link
→ 같은 inode를 공유하는
한 가지 동일한 파일

symbolic link
→ 다른 위치에 링크

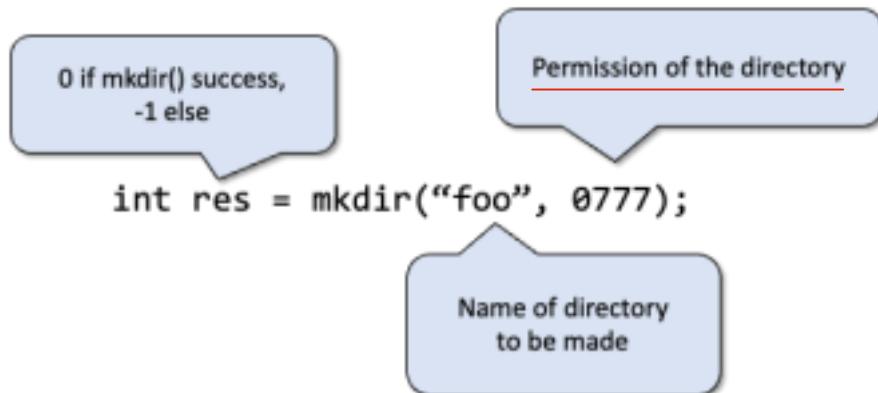
inode 구조 예시 { i_num } ⇒ inode 번호



(11) Making Directories

mkdir() system call : 디렉토리 생성

- Created directory contains two (user-readable name, low-level name) pairs: current("."), parent directory(..)



(12) Reading Directories

opendir(), readdir(), closedir() system calls

- readdir() returns information about its entry in struct dirent

- DIR

- FILE fopen()

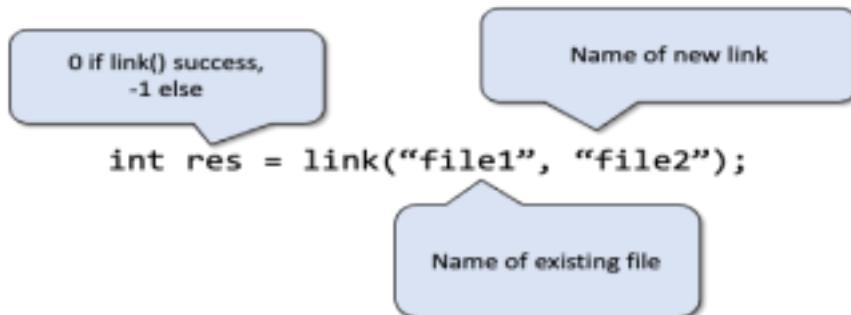
```
struct dirent {  
    ino_t      d_ino; /* inode number */  
    off_t      d_off; /* offset to the next dirent */  
    unsigned short d_reclen; /* length of this record */  
    unsigned char d_type; /* type of file; not supported  
                           by all file system types */  
    char       d_name[256]; /* filename */  
};
```

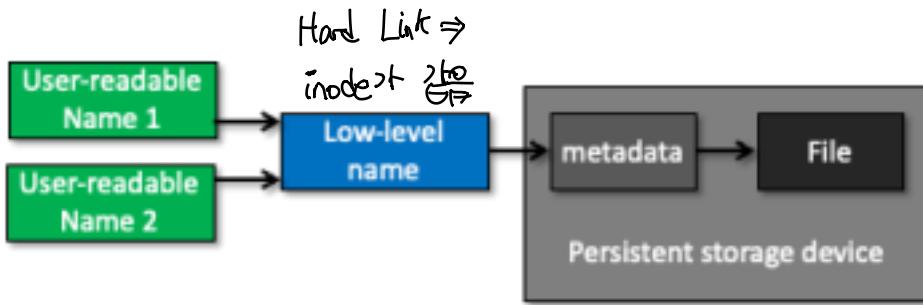
(13) Hard Links

- link() system call : 동일한 파일에 대한 새로운 (user-readable name, low-level name) 쌍을 생성

- The file could be accessed using two user-readable name

- link() increases number of hard links of the file





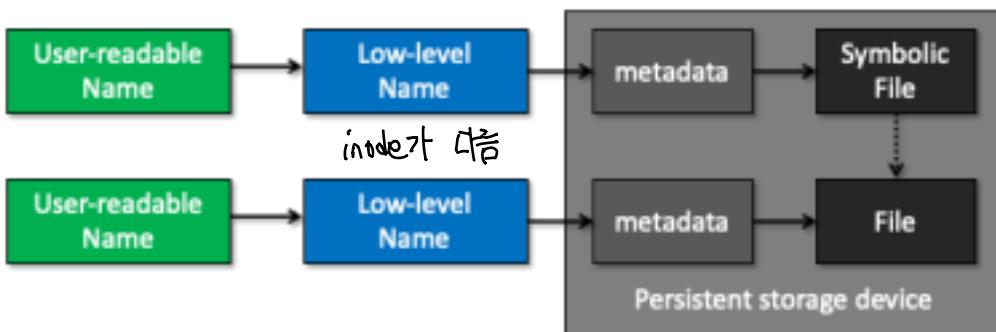
(14) Deleting Files

- Unlink() system call : 디렉토리에서 (user-readable name, low-level name) 쌍 삭제하고 파일의 링크 수를 줄임
 - If number of hard links became 0, file system free inode and related data blocks from the storage device



(15) Symbolic Links ≈ 매향기, 단축 아이콘

- Hard links could not be made for Directory (cycle could be created) or Files in other disk partitions (Low-level numbers are only unique within a particular file system)
- symbolic link is special type file that points other file or directory
 - Unlike hard link, dangling reference could be made



\$ls -i

```
$ echo 'hello' > hello
$ ln -s hello hello.symlink
$ ls -l
total 4
-rw-rw-r-- 1 user user 6 Jun  3 04:00 hello
[1]rwxrwxrwx 1 user user 5 Jun  3 04:00 hello.symlink -> hello
$ cat hello
$
hello
$ cat hello.symlink
hello
$ rm hello
$ cat hello.symlink
cat: hello.symlink: No such file or directory
```

Symbolic link

inode 다른

가리키는 파일이 없어짐

(16) Mounting a File System

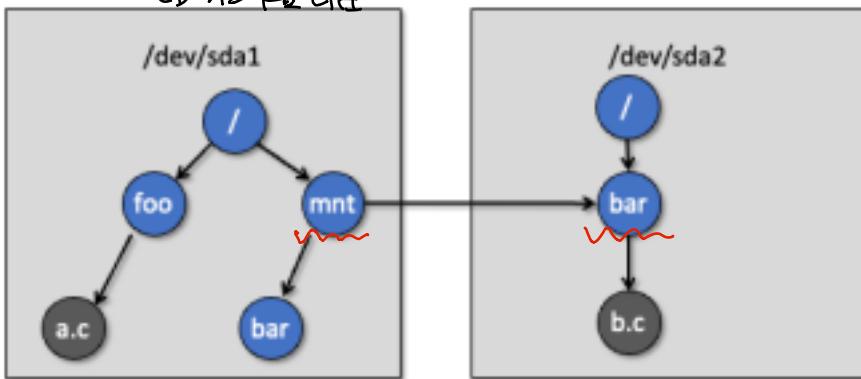
Tool mount connects root directory of a file system into a directory of other file system

- Every file in every file system could be accessed uniformly

- Mount -t (파일시스템 타입) < 장치 파일 > <마운트 포인터>

```
$ mount -t ext4 /dev/sda2 /mnt  
$ mount -t iso9660 /dev/cdrom /media/cdrom
```

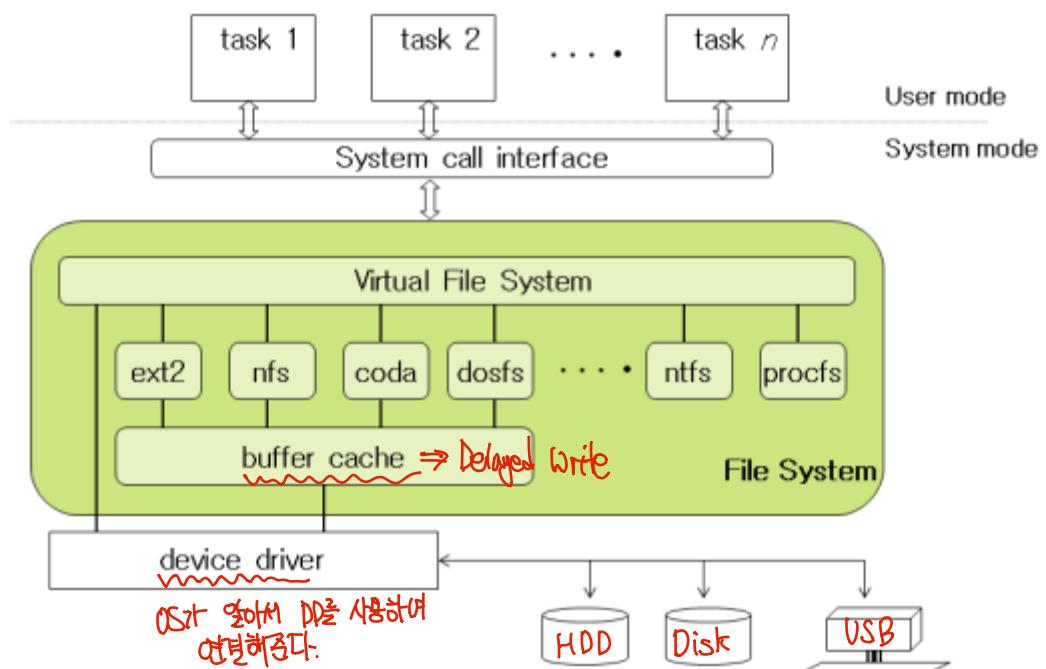
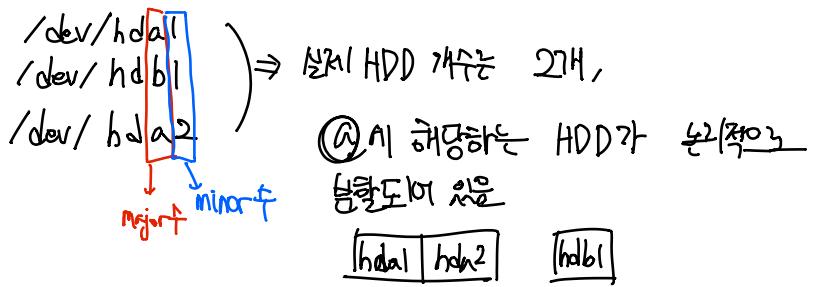
CD 파일시스템 마운트



File Management & File Systems

(1) 파일 시스템 전체 구조

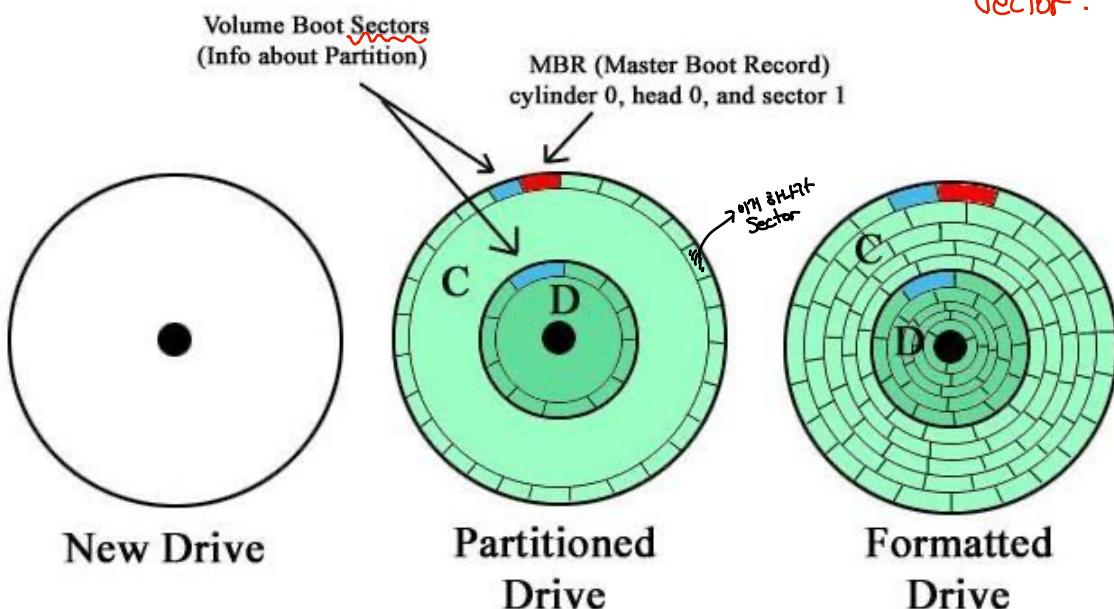
- HDD / SSD → File System
 - ✓ 데이터 저장과 추출을 어떻게 하느냐를 제어
 - ✓ 하드웨어의 기능을 추가하지 않은 순수 SW
 - 유연성이 아주 높음
 - 데이터 구조가 다른 형태가 아주 많으나 호환성이 높음파일시스템
 - 컴퓨터에서 파일이나 자료를 쉽게 발견 및 접근할 수 있도록 보관 또는 조직하는 체계
 - 파일 시스템은 일반적으로 크기가 일정한 블록들의 배열(섹터, 512B ~ 16KB)에 접근할 수 있는 자료 보관 장치 위에 생성되어 이러한 배열들을 조직
 - 자료 = '클러스터' = '블록' => 파일 하나가 필요로 하는 디스크의 최소 공간



(2) 디스크

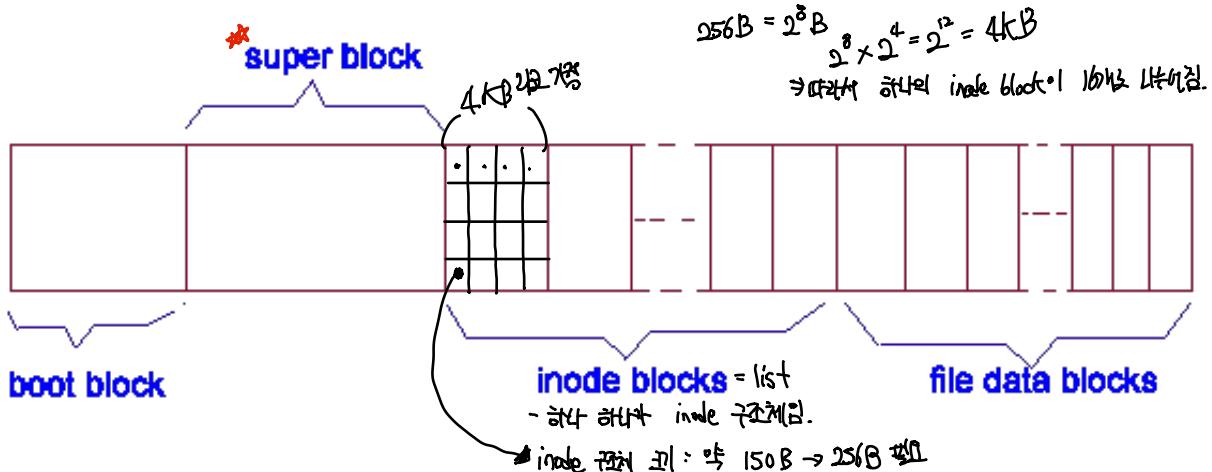
- 디스크 파티션 → **file system** 시스템 풀

Sector:



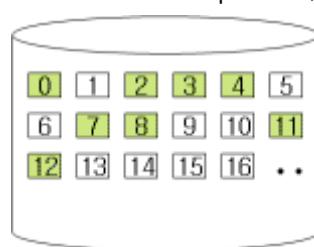
4KB
↑
하나의 Block은 여전히의 Sector 3
이기지 않는다.
512B
↑

- 디스크 블록의 크기와 디스크 파티션
- 통상 파일시스템 블록 크기와 동일하지만 때때로 그 보다 클 수 있음
 - ✓ 전통적인 UNIX : 512B → 1KB → 4KB → 현재 OS에서는 점점 큰 Block size를 사용하고 있다.
 - ✓ BSD : 4096KB → 8KB → 16KB
- 디스크 상에는 파일시스템에 존재하는 모든 파일들의 inode를 내포한 "inode 테이블" 또는 "inode 리스트"가 있음
 - ✓ 파일이 오픈되면 그 파일의 inode가 주기억장치로 반입되어 "메모리 내 inode 테이블"에 저장됨
 - ✓ di_inode vs incore_inode

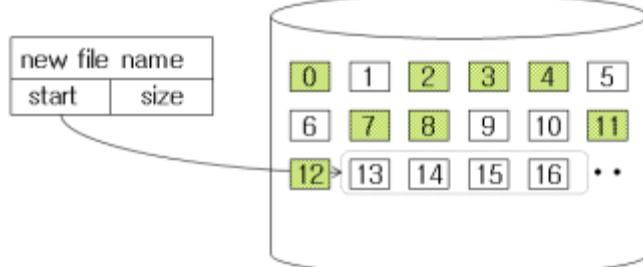


System V disk Partition Layout

- 디스크 블록 관리
- 시나리오
 - ✓ 새로운 14KB 크기의 파일을 디스크에게 쓰려고 함
 - ✓ 디스크 블록의 크기는 4KB로 가정 (따라서 4개의 디스크 블록이 필요)
 - ✓ 아래 그림에서 빛금이 있는 디스크 블록들은 이미 사용 중인 (파일의 데이터를 갖고 있는) 블록이라고 가정
 - ✓ 어떤 디스크 블록들을 할당할 것인가? (disk block allocation problem)



- 디스크 블록 할당 방법
 - 연속 할당 방법 (sequential allocation) 12가지 가능하지만 행성됨 ⇒ 여기 기준으로 할당 시작

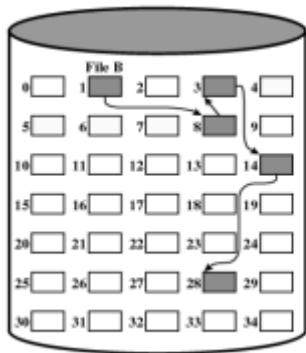
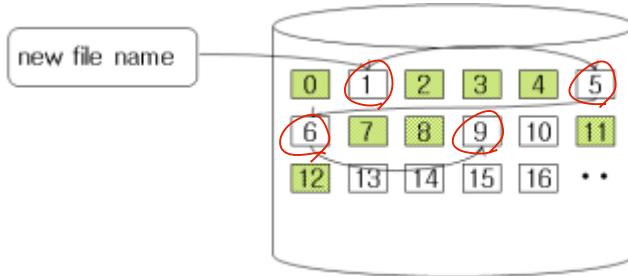


쓰지 않는 이유
→ 중간에 할당 해제된 경우
쓰지 않는 부분이 끊어 날아가
계속 양쪽을 해주어야 한다.

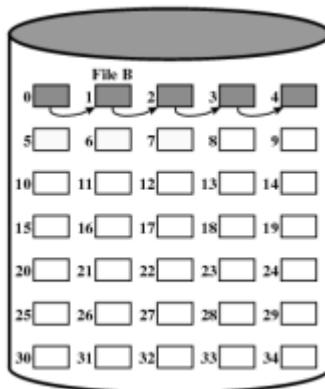
- 불연속 할당 방법 (non sequential allocation)
 - ✓ 블록 체인(block Chain) 방법 : 빈어있는 블록을 연결해서 관리하자!!

쓰지 않는 블록
→ 포함된 관계에 필요한 블록
한정의 불편함

- 현재 시나리오에서는 1, 5, 6, 9 블록을 할당
- 할당된 블록을 링크로 연결



File Allocation Table		
File Name	Start Block	Length
File B	1	5
...
File B	0	5
...

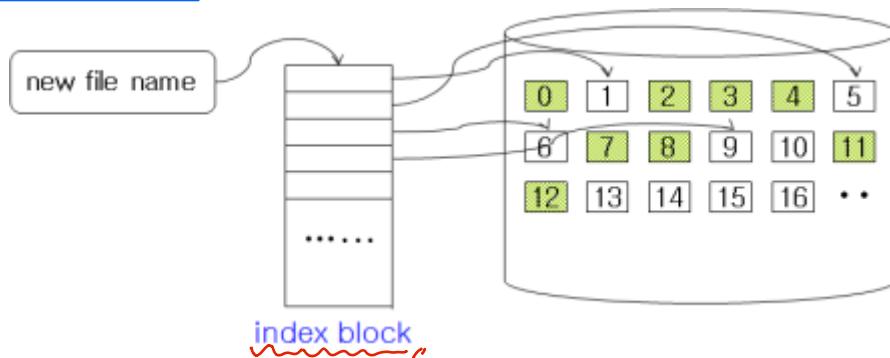


File Allocation Table		
File Name	Start Block	Length
File B	0	5
...
File B	0	5
...

⟨Compaction 후⟩

~~inode~~ **inode** → 현대 OS에서 사용하는 기법

인덱스 블록 (index block) 기법

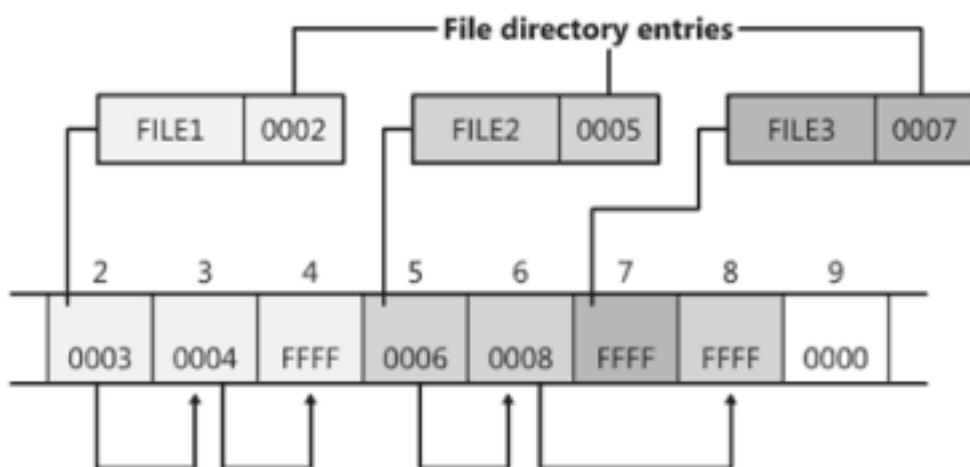
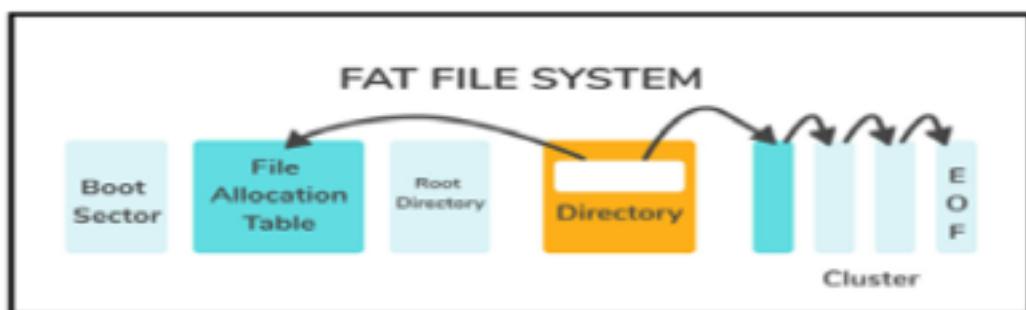


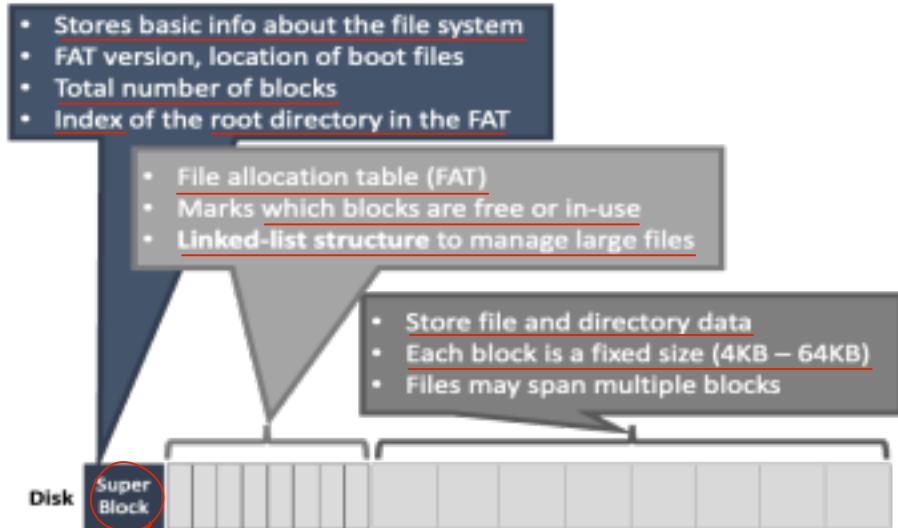
- ✓ 불연속 할당 방법 : FAT (File Allocation Table) 배열로 관리



인덱스

단점: 배열 관리 구현되어서 효율성이 떨어진다.





- Directories are special files
- ✓ File contains a list of entries inside the directory
- Possible values for FAT entries:
- ✓ 0 - entry is empty
- ✓ 1 - reserved by the OS
- ✓ $1 < N < 0xFFFF$ - next block in a chain
- ✓ 0xFFFF - end of a chain

- FAT 의 발전

- 1977 by MS From QDOS (Quick & Dirty OS)
- FAT12
 - ✓ Used on volumes smaller than 16MB (Floppies)
 - ✓ 클러스터 갯수 1~0xFFF
- FAT16
 - ✓ Used on volumes 16MB~512MB
 - ✓ 클러스터 갯수 0xFFF ~ 0xFFFF
- FAT32
 - ✓ Used on volumes 512MB~2TB
 - ✓ 클러스터 갯수 0xFFFF ~ 0xFFFFFFFF
- exFAT
 - ✓ NTFS와 호환용
- vFAT

Directory table entry (32B)

Filename {8B}
Extension {3B}
Attributes {1B}
Reserved {1B}
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)

File allocation table	
0	Volume info
1	Free
2	6
3	8
4	10
5	EOC
6	5
7	3
8	Free
9	11
10	EOC
11	...

※ 클러스터의 갯수 = 파티션의 크기 / 클러스터의 크기

- NTFS (New Technology File System)
- Master File Table 사용
- 관리 Mirror와 파일 로그 유지 -> 복구 가능
- 이론적 크기 16EB, 실제 크기 16TB



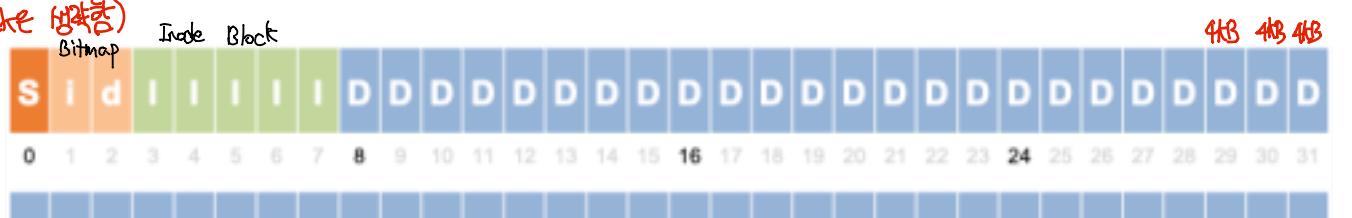
- 디스크 상 파일의 제어 블럭
 - FCB (File Control Block) : structure containing various pieces of file information.
 - ✓ Unix : inode (include/linux/fs.h), stat command / stat structure
 - ✓ Windows : MFT (Master File Table)
- 파일의 내용은 디스크의 하나 이상의(종종 많은) 블록에 저장
- FCB는 이러한 블록을 기록하고 액세스할 수 있는 방법 제공해야 함

~~※~~

간단한 파일시스템 구조의 예

- 디스크는 동일한 크기(4KB)의 블럭으로 나눔 FCB 첨부 FCB 주의 4KB
- 디스크에 64 개의 블럭이 있다고 가정
- ✓ 각 블록들은 특정 목적으로 분류됨
- ✓ User data blocks (56 개 블록) : D
- ✓ Key meta data blocks (5 개 블록) : inode 테이블의 Inode : I
- ✓ Allocation structure blocks (2 개 블록) : inode 및 데이터에 대한 두 개의 비트 맵 (free 0, used 1) : i, d
- ✓ Superblock : 파일 시스템의 전체 정보 : S

(Block block) (상자)



32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

S : Super block

I : Inode table

D : Data block

i : Inode bitmap

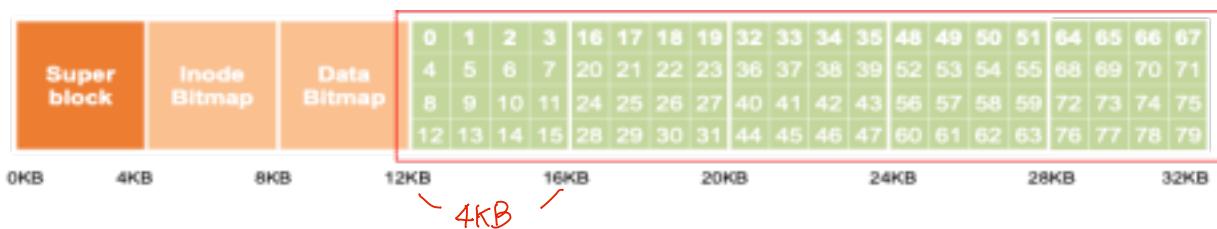
bitmap 값이 0이다
→ Inode table의
해당 block을 사용하지 않을

d : Data bitmap

데이터 블록을 사용하는데 대한
여부
0 or 1
(이다 → 사용함)

- Unix의 파일 관리
 - 파일의 종류
 - ✓ 일반파일(Regular or ordinary)
 - 0개 이상의 데이터 블록에 저장된 임의의 데이터를 포함
 - 일반파일에는 사용자, 응용 프로그램, 또는 시스템 유ти리티 프로그램이 만들어 입력한 정보가 있음
 - 파일시스템은 일반파일에 대해 어떠한 내부 구조도 부과하지 않고 단지 바이트 스트림으로 취급
 - ✓ 디렉터리(Directory)
 - 파일명과 inode(index node) 번호들의 리스트
 - 디렉터리들은 그림 12.4에 보인 것처럼 계층적으로 구성
 - 디렉터리 파일은 실질적으로 특수한 쓰기권한을 가진 일반파일로, 오직 파일시스템만 쓸 수 있고 읽기접근은 사용자 프로그램에 서도 가능
 - ✓ 특수파일(Special)
 - 데이터는 없지만, 물리적 장치들을 파일명에 사상시키는 기법을 제공
 - 파일명은 주변장치, 예를 들면 단말기나 프린터와 같은 장치를 접근하는데 사용
 - 각 입출력 장치는 특수파일과 연계
 - ✓ 명명파이프(Named pipes)
 - 파이프는 프로세스 간 통신장치
 - 파이프 파일은 파이프의 출력을 읽는 프로세스가 선입선출 기반으로 데이터를 받을 수 있도록 입력된 데이터를 버퍼시킴
 - ✓ 링크(Links)
 - 링크는 본질상 존재하는 파일에 대한 또 다른 파일명
 - ✓ 심볼릭 링크(Symbolic links)
 - 자신과 연결된 파일명을 저장하고 있는 데이터 파일이다.

- inode의 정의
 - ✓ 파일에 대한 메타데이터



- ✓ 현대 UNIX 운영체제는 복수 파일 시스템을 지원하지만 이 모든 파일 시스템을 파일 시스템을 지원하고 디스크를 파일에 할당하기 위한 동일한 기본 시스템으로 사상
- ✓ 모든 유형의 UNIX 파일은 inode를 통해 운영체제가 관리
- ✓ UNIX/Linux에서 파일을 관리하기 위한 객체
- ✓ inode는 각각의 파일에 대한 주요 정보가 저장된 제어 구조
- ✓ 몇몇의 파일명이 하나의 inode와 관련될 수 있지만, 활성화된 inode는 단 하나의 파일과 연계되며 각 파일은 단 하나의 inode에 의해 제어됨
- ✓ 파일이 새로 생성되면 만들어짐
- ✓ 파일의 모든 정보를 관리
- ✓ 파일에 속한 블록 위치 (index block 방법과 유사)
- ✓ 파일 소유자 및 접근 권한
 - 파일 시간 정보

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i.osd2	another OS-dependent field

Figure 40.1: The Ext2 Inode

· 파일 유형 : 커널은 정규 파일 뿐만 아니라 디렉터리, 디바이스, 파일, 소켓 등도 파일이라는 추상화 객체로 관리

✓ Disk에 정적으로 존재

✓ inode의 위치 계산

· blk = (inode * sizeof(inode_t)) / blockSize; \Rightarrow Block 위치

· sector = ((blk * blockSize) + inodeStartAddr) / sectorSize; \Rightarrow Sector 위치

$$\frac{32 \times 256B}{2^4 \cdot 2^{12}B} = 2^8 = 256 \text{ 블록에 위치}$$

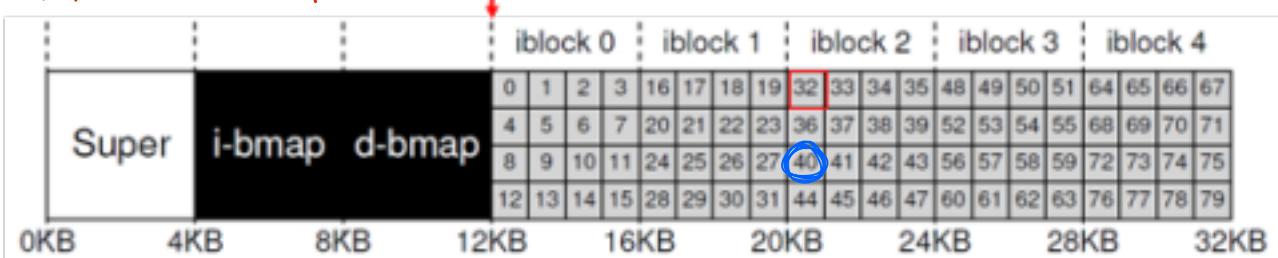
$$\frac{2 \times 4KB + 12KB}{512B} = \frac{20KB}{512B} = 40 \text{ 블록에 위치}$$

Inode number	32
Inode size	256 B
Inode start address	12 KB
Block size	4 KB
Sector size	512 B

↳ 예제는 8개임을 알 수 있음.

$$\begin{aligned} \text{iblock} &= \frac{(\text{i_number} * \text{inode_size}) / \text{block_size}}{2^4} \\ &= 32 * 256B / 4KB \\ &= 2 \rightarrow 256\text{개 INode Block에 존재한다.} \\ \text{sector} &= \frac{((\text{iblock} * \text{block_size}) + \text{i_start_addr}) / \text{sector_size}}{512B} \\ &= (2 * 4KB + 12KB) / 512B \\ &= 40 \end{aligned}$$

→ 40번재 Sector의 위치함



- 파일의 유형 및 접근 모드

✓ 파일의 소유주 및 그룹 접근 식별자

✓ 파일 생성시간, 가장 최근에 읽히거나 쓰인 시간, 그리고 inode가 시스템에 의해 가장 최근에 갱신된 시간

✓ 파일 크기(바이트 단위)

✓ 블록 포인터의 나열, 다음 절에서 설명됨

✓ 파일에 의해 사용 중인 물리 블록의 개수(간접지정 포인터와 속성을 저장하기 위해 사용 중인 블록 포함)

✓ 파일을 가리키는 디렉터리 항목의 개수

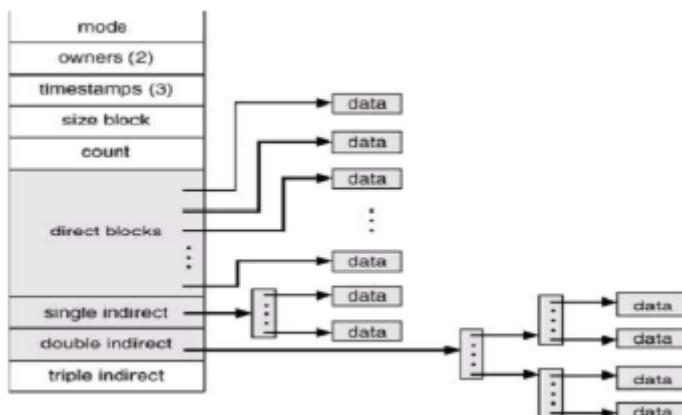
✓ 커널과 사용자에 의해 세트 가능한 파일의 특성을 나타내는 플래그

✓ 파일의 생성 번호(inode가 새 파일에 할당될 때마다 배정되는 임의 선정 번호; 생성 번호는 삭제된 파일을 참조하는 inode를 검출하기 위해 사용)

- ✓ inode에 의해 참조되는 데이터 블록의 블록크기(보통 파일 시스템 블록 크기와 같지만 때로 더 클 수 있음)
- ✓ 확장된 속성 정보의 크기
- ✓ 0 또는 그 이상의 확장 속성 항목

- 블록의 크기

- ✓ 통상 파일시스템 블록크기와 동일하지만 때때로 그 보다 클 수 있음
- ✓ 전통적인 UNIX 시스템에서는 512 바이트의 고정 값이 사용
- ✓ FreeBSD의 블록크기의 최솟값은 4096바이트(4Kbyte)
 - 블록크기는 4096보다 크거나 같은 임의의 2의 몇승 값을 사용할 수 있음
- ✓ 일반적인 파일 시스템의 경우 블록크기로 8Kbyte 또는 16Kbyte가 사용
- ✓ 디스크 상에는 파일시스템에 존재하는 모든 파일들의 inode를 내포한 “inode 테이블” 또는 “inode 리스트”가 있음
 - 파일이 오픈되면 그 파일의 inode가 주기억장치로 반입되어 “메모리 내 inode 테이블”에 저장됨



File Mode	16-bit flag that stores access and execution permissions associated with the file.
	12-14 File type (regular, directory, character or block special, FIFO pipe 9-11 Execution flags 8 Owner read permission 7 Owner write permission 6 Owner execute permission 5 Group read permission 4 Group write permission 3 Group execute permission 2 Other read permission 1 Other write permission 0 Other execute permission
Link Count	Number of directory references to this inode
Owner ID	Individual owner of file
Group ID	Group owner associated with this file
File Size	Number of bytes in file
File Addresses	39 bytes of address information
Last Accessed	Time of last file access
Last Modified	Time of last file modification
Inode Modified	Time of last inode modification

- 큰 파일 지원

- ✓ 직접 포인터으로 지정할 수 있는 데이터 블럭 수의 한계 (Unix 10, Linux 12)
- ✓ UNIX 시스템들은 10개 내외 직접 포인터로 데이터 블럭을 가르킴
- ✓ 다중 인덱스 기법 사용
 - 간접 포인터 기법
 - 직접 (direct pointers)

- 간접/이중간접/삼중간접 포인터 (single/double/triple indirect pointers)
- 간접 포인터가 가르키는 곳을 따라가면.. 디스크 블럭이 있는데… 데이터 없고 4바이트 짜리 1024.. 포인터가 4KB의 데이터 블럭 가르킴
- 데이터블럭이 1 KB : 4바이트 256 개의 포인터

내 파일의
1324724639 B 일때
약 1.2GB 일때

A 파일 구조에 이 파일이
들어갈 수 있을까?

1724639 B 일때

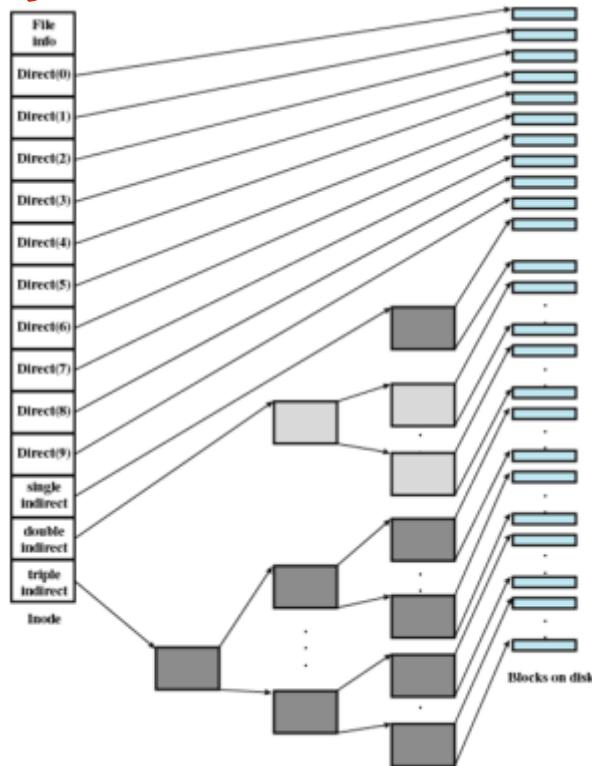
A의 파일 구조에 삼중간접포인터가

존재하면

몇개의 삼중 간접 포인터가 필요?

(이전 직접, 간접, 이중간접은 빼고
나머지 사용에 대해서 계산)

$$2^{10} \times 2^6$$



A. 데이터블록 크기가 1KB 일 때

$$\begin{aligned} \text{직접: } 114 &= 5 \times 1K = 5KB \\ \text{간접: } 1M &= 1 \times 256M \times 1KB = 256KB \\ \text{이중간접: } 114 &\times 2^6 \times 2^6 = 2^{16} KB \\ &\text{하나의 파일의 가치는 } 2^{16} KB \\ &\therefore \text{최대 } 5KB + 256KB + 2^{16} KB \end{aligned}$$

데이터블록 크기가 512B 일 때

$$\textcircled{1} \text{ 직접: } 114 \rightarrow 512B \times 11$$

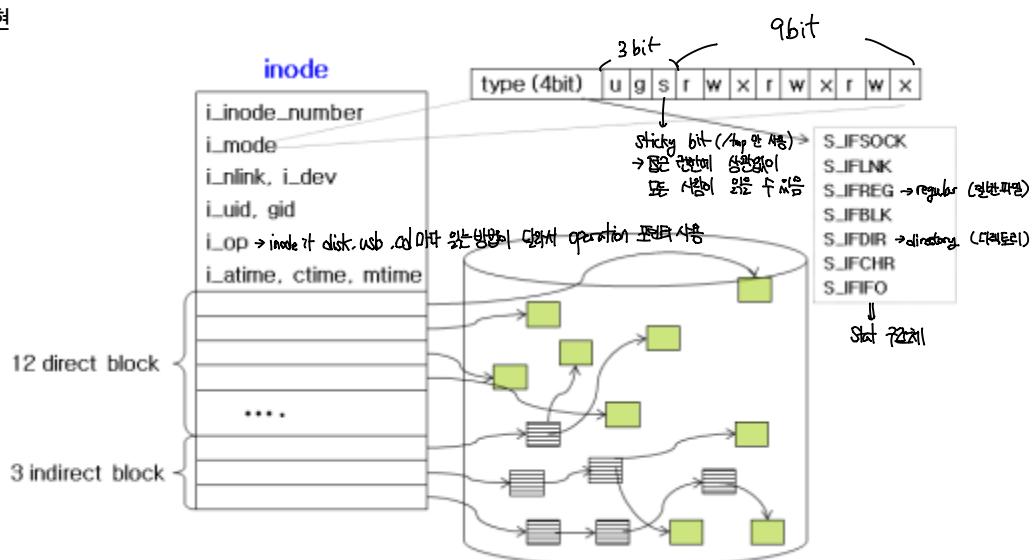
$$\textcircled{2} \text{ 간접: } 114 \rightarrow 1 \times 256M \times 512B$$

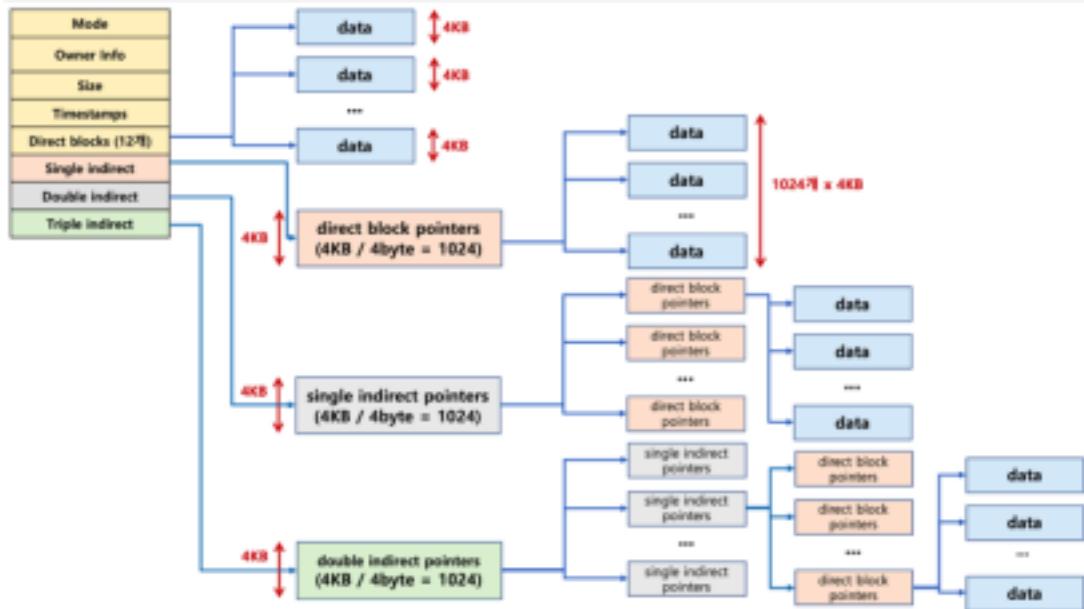
$$\textcircled{3} \text{ 이중간접: } 1 \rightarrow 1 \times 2^n \times 2^n \times 512B$$

$\textcircled{1} + \textcircled{2} + \textcircled{3}$ 하면

하나의 파일의 가치는 최대
파일 크기가 나온다.

- 리눅스에서 구현





- Directory Organization

- Directory

- ✓ 특수 형태의 파일
- ✓ 디렉토리는 하나의 inode와 데이터 블럭(들)을 가지고 있음
- ✓ 디렉토리 항목들은 선형리스트 형태로 구현
- ✓ 성능을 위해 보다 복잡하게 구현된 경우가 많음
 - XFS : B-tree, 파일 생성이 빠르며, 전체 디렉토리를 탐색하지 않아도 됨

dirent.h

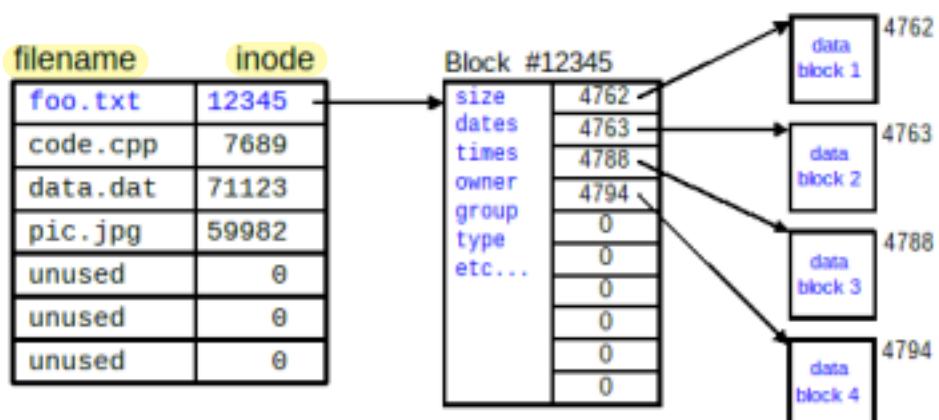
```

Int di_inode;
char d_name[];
...
int closedir(DIR *);
DIR *opendir(const char *);
struct dirent *readdir(DIR *);
int readdir_r(DIR *, struct dirent *, struct dirent **);
void rewinddir(DIR *);
void seekdir(DIR *, long int);
long int telldir(DIR *);

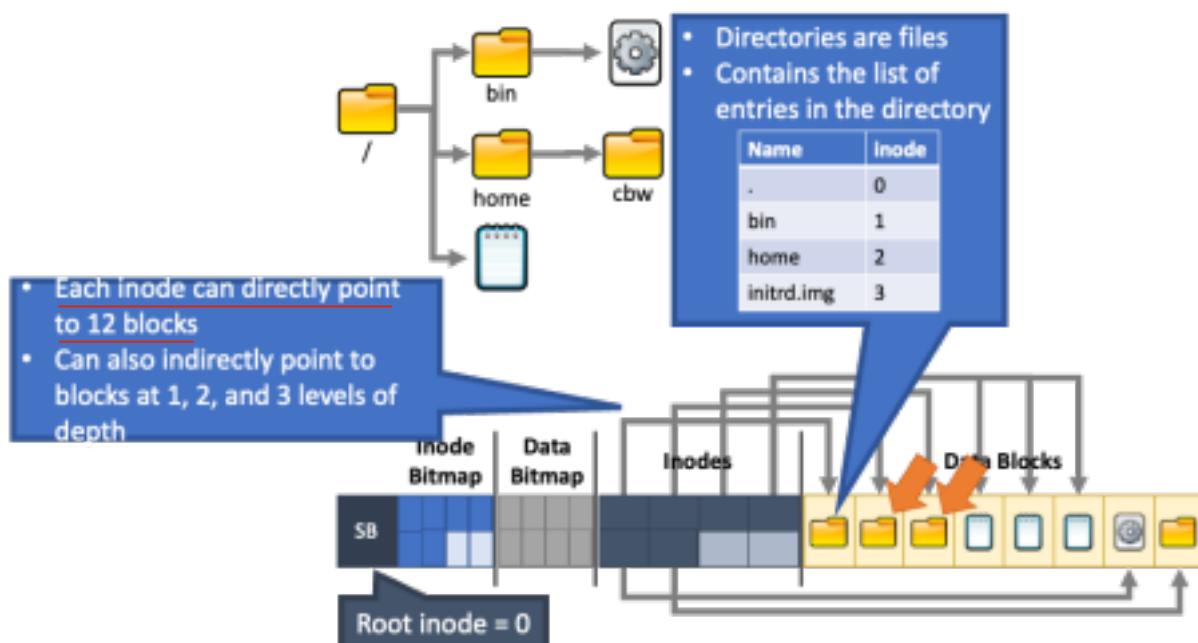
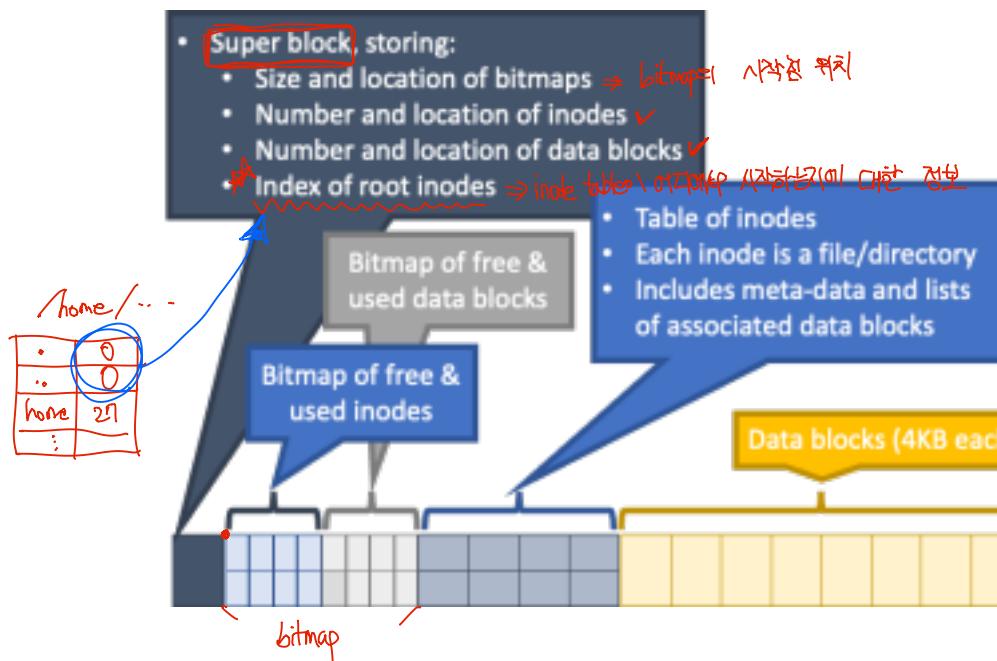
```

inum	recflen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

directory entries inode data blocks



- UFS의 디스크 파티션



- Free Space 관리
 - 새 파일 또는 디렉터리를 할당하려면
 - ✓ 사용 가능한 inode 및 데이터 블록 추적
 - inode 비트 맵과 데이터 비트 맵을 사용
 - ☞ Free list나 B 트리 등도 사용



- 새로운 파일 할당을 위한 정책 고려
 - ✓ 사용 가능한 블록 시퀀스를 찾아 새로운 파일에 할당
 - ✓ 파일의 일부를 디스크에 순차적으로 배치하여 성능 향상
 - ✓