File System Implementation

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

The Way to Think

파일 시스템을 어떻게 구현?

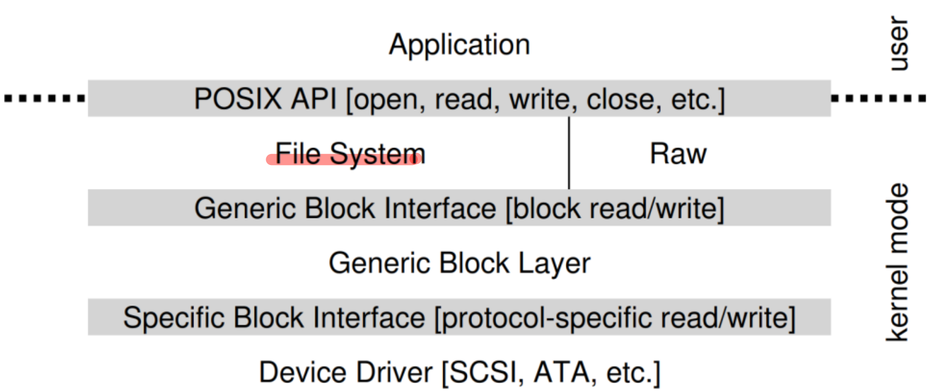
- **We introduce a simple file system implementation vsfs**
  - Specifically, vsfs (Very Simple File System) will be implemented
  - The file system is pure software; no hardware feature is required
    순수 소프트웨어.
- **We need to understand how the file system works in two aspects**

  → 자료구조 (파일들을 저장하는) / array + tree, ... (데이터와 메타데이터를 저장하기 위해)

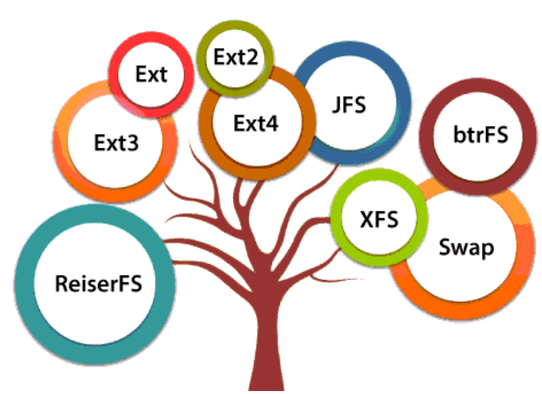
  1) The first is the data structures of the file system
     In other words, what types of on-disk structures are utilized by the file system to organize its data and metadata? → array, tree, etc

  접근 방법 → 자료구조에 어떻게 접근할 것인가 / open(), read(), write()의 프로세스

  2) The second aspect of a file system is its access methods
     How does it map the calls made by a process, such as `open()`, `read()`, `write()`, etc., onto its structures?
     의 호출은 어떻게 처리할지.



Types of Linux File System

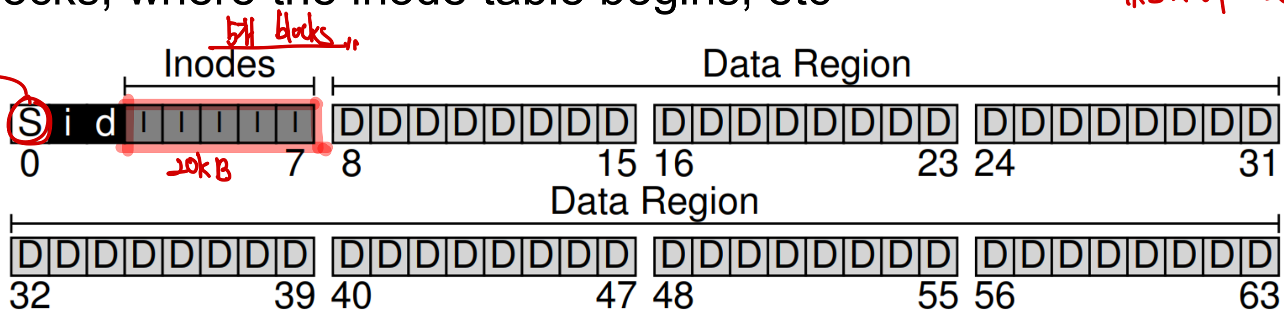


Overall Organization

- **The first thing we'll need to do is divide the disk into block**
  - Simple file systems use just one block size, commonly-used size of 4KB
  - Our view of the disk partition is a series of blocks (0~N-1); Assume 64 blocks
- **Think about what we store in these blocks to build vsfs**
  - To store user data, the data region of 56 blocks on the disk is reserved →저장된 사용자 데이터.
  - To store metadata, which contains (file size, owner, etc), file systems have inode; 5 blocks of inode table; 256 byte per inode, 20KB can hold 80 total inodes →파일크기, 접근권한, 접근, 수정시간, ...
  - Allocation structures to track if a block is free are required in file systems
  - Allocation-tracking methods: free list and bitmap, etc →그러한 공간에 inode, data를 저장할 수 있는지.
    bitmap is simple and thus vsfs will use bitmap for inode and data
  - The last block is reserved for superblock, which contains how many inodes and data blocks, where the inode table begins, etc →여기 배열.

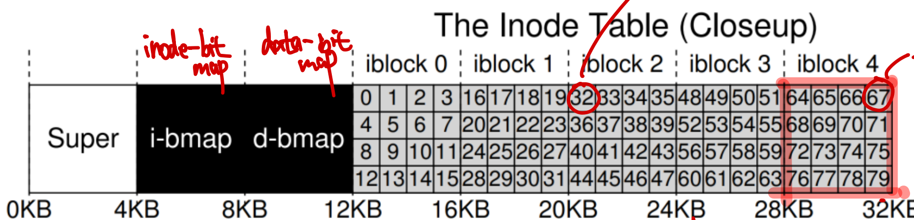
4KB x 64 = 256KB

file 시스템 전체와 관련된 정보.



File Organization: The Inode

- **One of the most important on-disk structures of file system is the inode, which is short for index node**
  - Each inode is implicitly referred to by a number (i-number; low-level name)



슈퍼블록에서 inode table의 시작거리. = 12KB

그런데 inode 32번이므로. (256B) x 32 = 8KB

20KB가 inode의 위치
↓
여기 가면 실제 data의 위치가 나오게 되고 위치로 가서 데이터를 찾는다.

  - The inode location can be calculated using the i-number
    e.g.) i-number: 32 → 32 × sizeof(inode) + 12KB = 32 × 256B+12KB=20KB
    To find the corresponding sector 20KB/512 = 40 =메타 데이터.
  - A general equation to find the sector:

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

  - Each inode includes the file's metadata, such as its size, protection, time, etc (e.g. simplified ext2 inode on right figure)

inode에 있는 정보 = file 메타와 대응된다.

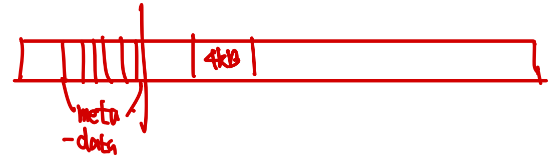
EX2

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

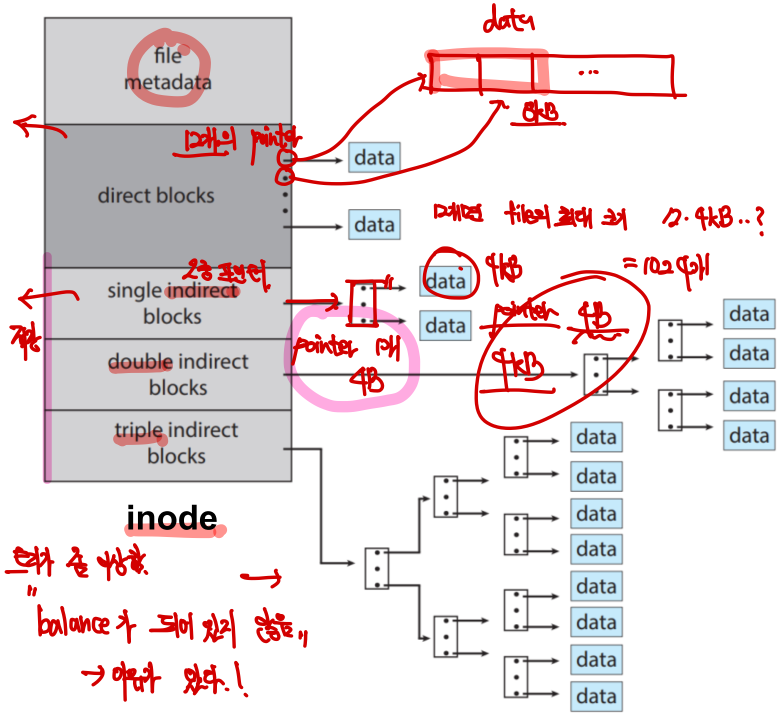
4x15=60

실제 데이터의 위치

The Multi-Level Index



- **To support bigger files, file systems uses an indirect pointer**
  - Instead of directly pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data
  - Each inode of vsfs contains 12 direct pointers and 1 indirect pointers; under 4-byte disk address, the file can grow to be $(12+1024) \times 4KB = 4144KB$
    1 indirect pointers
  - To have even larger files, we can add another pointer to inode: double indirect pointer → $(12+1024+1024^2) \times 4KB \approx 4GB$
    Double indirect 1개
  - Even more? triple indirect pointer
  - This imbalanced tree is referred to as the multi-level index approach
  - Why use an imbalanced tree like this? One finding is that most files are small



Most files are small	˜2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ˜50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Directory Organization & Free Space Management

어떤 inode와 데이터 블록이 사용가능한지에 대해 관리하고 이는 bitmap에 저장

- **Directories basically contain a list of (entry name, i-number) pair**
  - e.g.) a directory (i-number 5) has files (foo, bar, foobar_is_a_pretty_longname) with inode number 12, 13, and 24, respectively

i-node. meta.

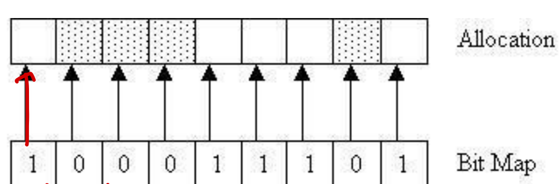
```
inum | reclen | strlen | name
 5      12        2      .
 2      12        3      ..
12      12        4      foo
13      12        4      bar
24      36       28      foobar_is_a_pretty_longname
```

구하고 보는 것.

  - Each entry has an inode number, record length (total bytes for name + any left over space), string length (name's actual length), and the file name

- **Free space management is important for all file systems**
  - 여유 공간은 bitmap으로 관리
  - vsfs uses two bitmaps to track which inodes and data blocks are free
  - When creating a file, the file system allocates the an inode and search through bitmap for a free inode, and allocate it to file; similar activity for data blocks

inode 관리, data 관리 2개 다 찾아야 한다.



새로 할당 가능

Access Paths: Reading and Writing

3개의 디스크 블록

- **Opening, reading, closing a file `/foo/bar` whose size is 12KB**
  - The file system finds the inode for **bar** from root (its i-number is already known)
  - **open()**: root inode (pointer) → root directory data (**foo** i-number) → **foo** inode (pointer) → **foo** directory data (**bar** i-number) → **bar** inode
  - **read()**: read **bar** data → update the inode (last access time, etc)
  - **close()**: close the file and deallocate the file descriptor

보통 2번

root directory (inode)

foo

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			① read							
						② read				
				③ read						
							④ read			
					⑤ read					
read()					⑥ read			⑦ read		
					⑧ write					
read()					read				read	
					write					
read()					read					read
					write					

① open: root directory의 inode로 가서 파일에 대한 경로를 찾아냄

② root 디렉토리의 data 부분에서 foo라는 파일, inode 번호를 찾고, 해당 inode를 가지고 foo 디렉토리 inode로 이동

③ foo 디렉토리 inode를 읽어 foo 디렉토리의 data 부분으로 감

④ foo 디렉토리의 data부분을 읽어서 bar라는 파일의 inode 번호를 찾아내고 bar inode로 이동

⑤ bar inode로 들어오면 이후 read를 할 수 있음

bar를 가리키는 것이 최종 목적지. bar 파일의 데이터 블록으로 이동

3개의 block에 대해

meta.

업데이트는 접근시간 때문에

접근시간은 inode에 다시 저장된다.

bar

4KB

Writing A File To Disk

새로운 디스크 블록을 할당할 수도 있음

- **Writing to a file is a similar process with the read**
  - Unlike reading, writing allocates a block unless the block is being overwritten
  - **`create()`**: inode bitmap (free inode) → inode (initialization) → directory data (link filename to i-number) → directory inode (update)
  - **`write()`**: data bitmap (free block) → inode (update) → actual write
  - **`close()`**: close the file and deallocate the file descriptor
  - 12KB write to **`/foo/bar`**: 10 I/O for file create and 5 I/O for each block write

free block은 i-bitmap에서 찾아야함

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read write	read write	read	read write			

사용가능 inode 찾음

bar의 inode를 업데이트

foo의 수정 → 메타 정보 업데이트.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

반복 x3

3번 반복

Caching and Buffering

inode 정보를 다 disk에서 가져와 사용

- **Reading and writing files can be expensive due to many disk I/O**
  - With a long pathname (e.g., /1/2/3/ ... /100/file.txt), the file system would literally perform hundreds of reads just to open the file

    어떤 file에 대한 inode 정보를 cache.
  - Most file systems aggressively use memory (DRAM) to cache important blocks

    →버퍼링으로 쓰기 작업도 빨라짐 (I/O 성능 향상)
- **Early file systems introduced fixed-size cache for popular blocks**

  static partition
  - This static partitioning of memory can be wasteful (e.g. 10% of memory)

    →이러한 경우
  - Modern systems, in contrast, employ dynamic partitioning approach; The OS integrates virtual memory pages and file system pages into unified page cache

    paging
- **Imagine the file open example with caching**
  - Opening a file generate many disk I/O (very slow) to read inode, directory data
  - What if opening again the same file or files in the same directory? → cache hit (no slow disk I/O) → performance↑

    disk 거치지 않아도 되니까.
- **Consider the effect of caching on writes** →데이터 불일치 문제가 고려 되어야 한다.
  - Write buffering certainly has a number of performance benefits: 1) batch update, 2) write scheduling, 3) avoid write (e.g. temporary files)

    →전파될 단에서 필요한?