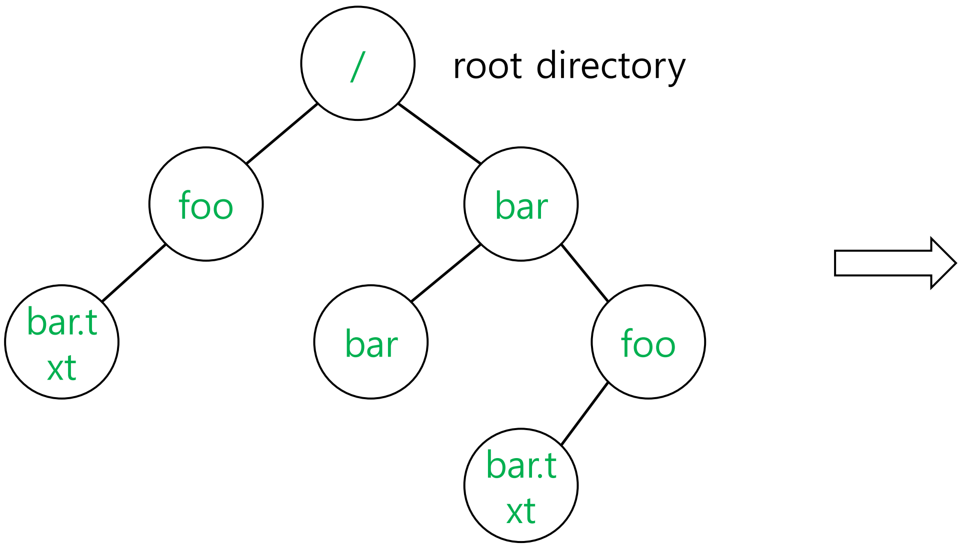
Operating Systems

Lecture 13

39. File and Directories

Concepts

- File
  - File is simply a linear array of bytes.
  - Each file has low-level name as 'inode number'
- Directory
  - A file
  - A list of <user-readable filename, low-level filename> pairs



An Example Directory Tree

vaild files :
/foo/bar.txt
/bar/foo/bar.txt

vaild directory :
/
/foo
/bar
/bar/bar
/bar/foo/

Interface: Creating a file

- Use `open` system call with `O_CREAT` flag.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, …);
```

  - `O_CREAT` : create file.
  - `O_WRONLY` : only write to that file while opened.
  - `O_TRUNC` : make the file size zero (remove any existing content).
- open system call returns file descriptor.
  - file descriptor is an integer, is used to access files.
  - Ex) `read (file descriptor)`
  - File descriptor table

```
struct proc {
...
  struct file *ofile[NOFILE]; // Open files
...
};
```

Interface: Reading and Writing Files

- An Example of reading and writing 'foo' file.

```
prompt> echo hello > foo   //save the output to the file foo
prompt> cat foo            //dump the contents to the screen
hello
prompt>
```

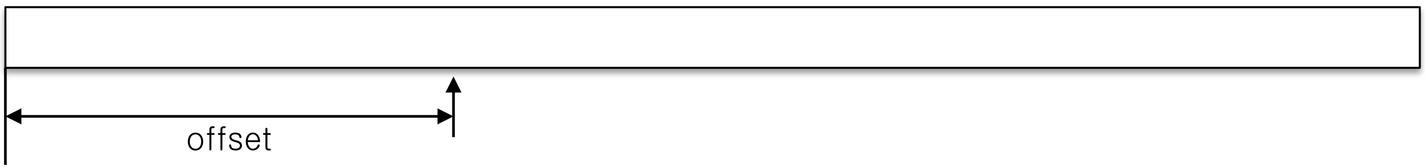
- The result of `strace` to figure out `cat` is doing.

```
prompt> strace cat foo   //strace to figure out what cat is doing
…
open("foo", O_RDONLY|O_LARGEFILE)  = 3
read(3, "hello\n", 4096)          = 6 /*the number of bytes to read*/
write(1, "hello\n", 6)            = 6
hello
..
..
..
prompt>
```

- `open()`: open file for reading with `O_RDOLY` and `O_LARGEFILE` flags.
  - returns file descriptor 3 (0,1,2, is for standard input/output/error)
- `read()`: read bytes from the file.
- `write()`: write buffer to standard output.

Reading and Writing Files (Cont.)

- OFFSET



  - The position of the file where we start read and write.
  - When a file is open, “an offset” is allocated.
  - Updated after read/write

- How to read or write to a specific offset within a file ?

```
off_t lseek(int fd, off_t offset /*location */, int whence);
```

  - Third argument is how the seek is performed.
    - `SEEK_SET` : to offset bytes.
    - `SEEK_CUR`: to its current location plus offset bytes.
    - `SEEK_END`: to the size of the file plus offset bytes.

abstractions

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
};
```

```
struct {
        struct spinlock lock;
        struct file file[NFILE];
} ftable;
```

Source code in xv6

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

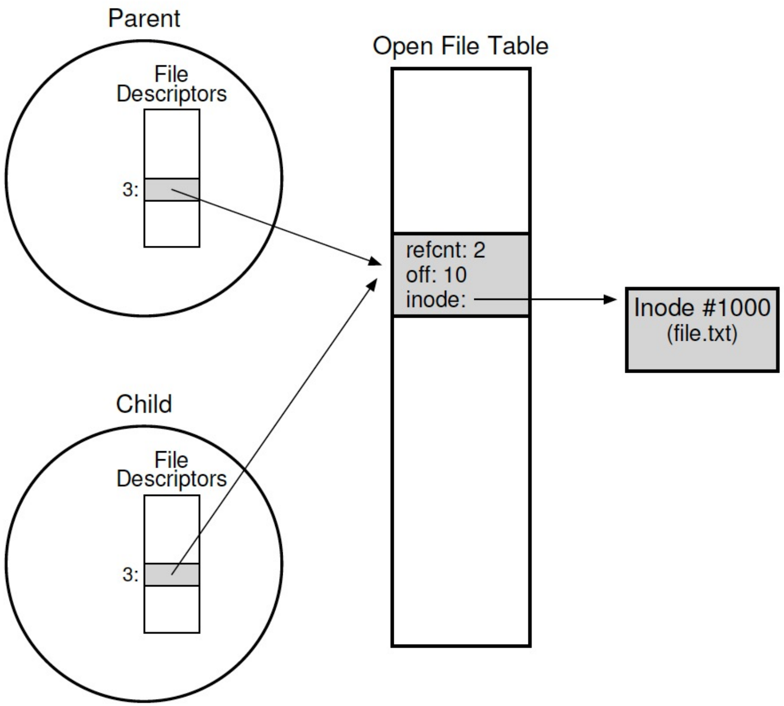
Sample traces

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

`fork()` and `dup()`

- Child process inherits the file descriptor table of the parent.



- Duplicating a file descriptor

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```

fsync()

- Persistency
  - write() : write data to the buffer. Later, save it to the storage.
  - some applications require more than eventual guarantee. Ex) DBMS
- fsync() : the writes are forced immediately to disk.

```
off_t fsync(int fd /*for the file referred to by the specified file*/)
```

- An Example of fsync().

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
int rc = write(fd, buffer, size);
rc = fsync(fd);
```

Renaming Files

- `rename()` :rename a file to different name.
  - It implemented as an atomic call.
  - Ex) change from foo to bar

```
promt > mv foo bar
```

- Saving a file in an editor

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Getting Information About Files

- `stat()` : Show the File metadata
  - metadata is information about each file, ex: size, permission, ..
- `stat` structure is below:

```
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 filesystem I/O */
    blkcnt_t st_blocks; /* number of 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 */
};
```

Getting Information About Files (Cont.)

- An example of `stat()`
  - All information is in a inode

```
prompt> echo hello > file
prompt> stat file

File: ‘file’
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Removing Files

- The result of `strace` to figure out what `rm` is doing.
  - `rm` is Linux command to remove a file
  - `rm` calls `unlink()` to remove a file.

```
prompt> strace rm foo
...
unlink("foo")
...
prompt>
```

Making Directories

- `mkdir()`: Make a directory
  - When a directory is created, it is empty.
  - Empty directory have two entries: . (itself), ..(parent)

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                          = 0
...
prompt>
```

```
prompt> ls -al
total 8
drwxr-x--- 2 roo root 6 Apr 30 16:17 ./
drwxr-x--- 26 root root 4096 Apr 30 16:17 ../
```

Reading Directories

- `readdir()`
  - Directory is a file, but with a specific structure.
  - When reading a directory, we use specific system call other than read().
  - A sample code to read directory entries.

```c
int main(int argc, char *argv[]) {
 DIR *dp = opendir("."); /* open current directory */
 assert(dp != NULL);
 struct dirent *d;
 while ((d = readdir(dp)) != NULL) { /* read one directory entry */
  printf("%d %s\n", (int) d->d_ino, d->d_name);
 }
 closedir(dp); /*close current directory */
 return 0;
}
```

Reading Directories

- Structure of the directory entry

```
struct dirent {
    char d_name[256]; /* filename */
    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 */
};
```

Deleting Directories

- `rmdir()`: Delete a directory.
  - `rmdir()` requires directory be empty before it deleted.
  - If you call `rmdir()` to a non-empty directory, it will fail.

Hard Links

- `link()`: Link old file and a new file.
  - Create hard link named file2. (link to inode not to a file)
  - Only can link to a file not to a directory

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2 /* create a hard link, link file to file2 */
prompt> cat file2
hello
```

- The result of `link()`
  - Two files have same inode number, but two different name(file, file2)

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

unlink Hard Links

- What `unlink()` is doing ?
  - Check reference count within the inode number.
  - Remove link between human-readable name and inode number.
  - Decrease reference count.
    - When only it reaches zero, It delete a file (free the inode and related blocks)

unlink Hard Links (Cont.)

- The result of `unlink()`

```
prompt> echo hello > file                  /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...          /* Link count is 1 */
prompt> ln file file2                     /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...          /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...          /* Link count is 2 */
prompt> ln file2 file3                    /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...          /* Link count is 3 */
prompt> rm file                           /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...          /* Link count is 2 */
prompt> rm file2                          /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...          /* Link count is 1 */
prompt> rm file3
```

Symbolic Links

- Symbolic link
  - Link to a file (not to inode)
  - Special file that contains path to the source file.
  - Hard Link cannot create to a directory.

- An example of symbolic link

```
prompt> echo hello > file
prompt> ln –s file file2  /* option –s : create a symbolic link, */
prompt> cat file2
hello
```

Symbolic Links (Cont.)

- Symbolic link is different file type.

```
prompt> ls -al
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../       /* directory */
-rw-r----- 1 remzi remzi 6 May 3 19:10 file          /* regular file */
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file /* symbolic link */
```

- Symbolic link is subject to the dangling reference.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Summary

- Create file
- read/write/lseek
- mkdir/readdir
- fsync
- hardlink/softlink

40. Filesystem Implementation

Overview

- In this chapter, we study very simple file system (vsfs)
  - Basic on-disk structures, access methods, and various policies of vsfs
- We will study…
  - How can we build a simple file system?
  - What structures are needed on the disk?
  - What do they need to track?
  - How are they accessed?

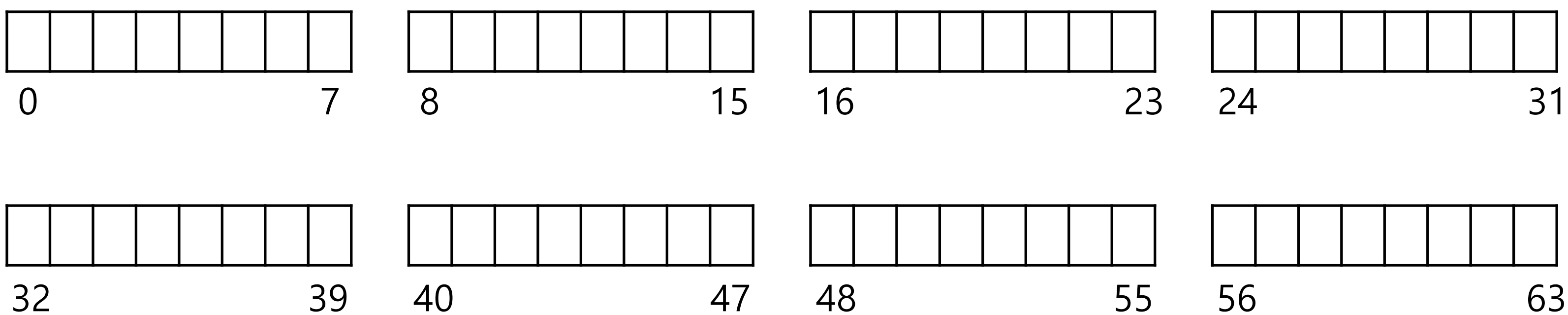
File system Implementation

- What types of data structures are utilized by the file system?
- How file system organize its data and metadata?
- Understand access methods of a file system.
  - `open()`, `read()`, `write()`, etc.

Overall Organization

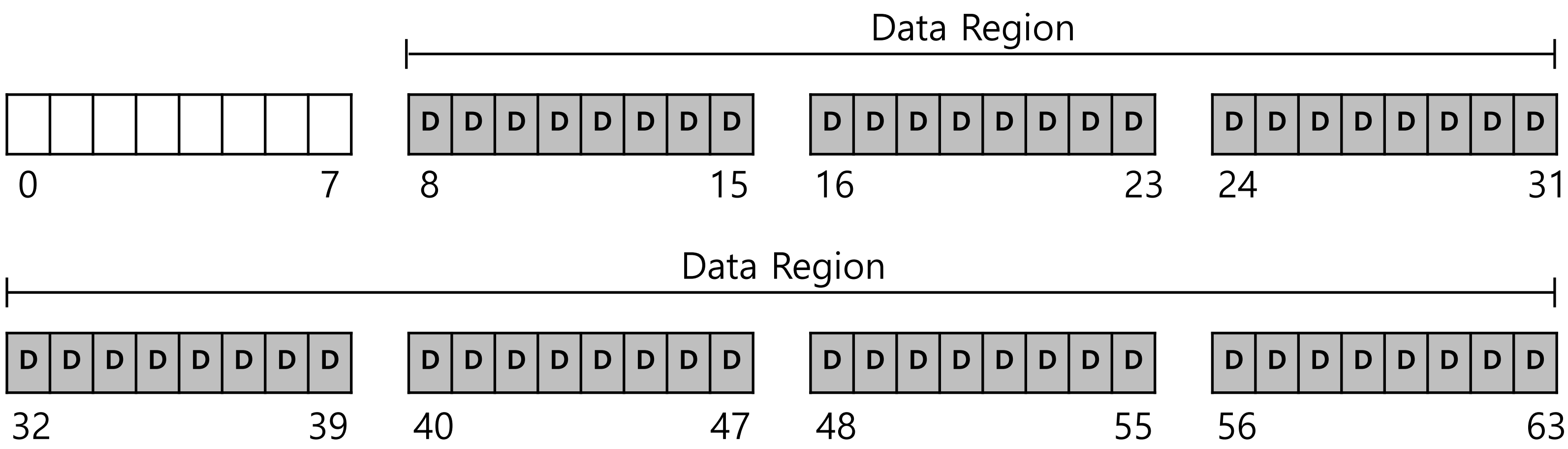
- Let's develop the overall organization of the file system data structure.

- Divide the disk into blocks.
  - Block size is 4 KB.
  - The blocks are addressed from `0 to N -1`.



Data region in file system

- Reserve data region to store user data

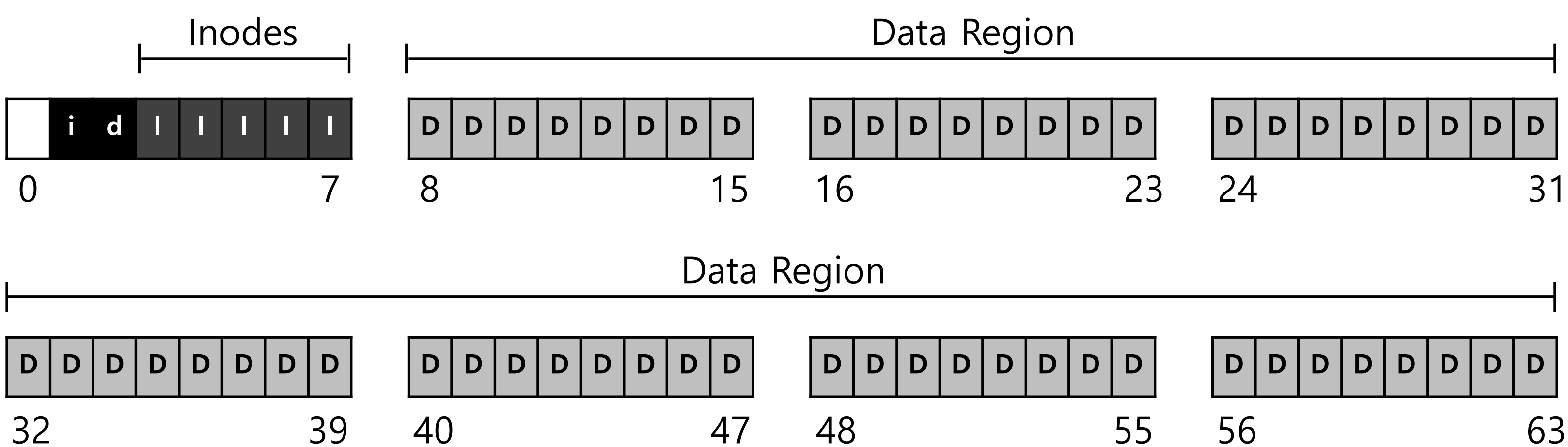


- File system has to track which data block comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

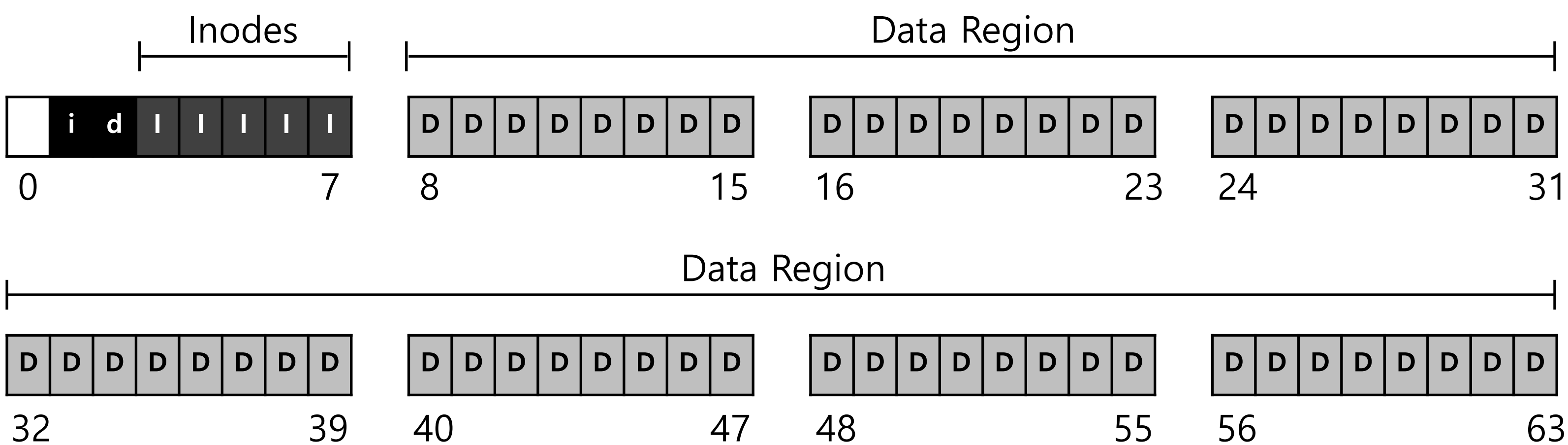
Inode table in file system

- Reserve some space for inode table
  - This holds an array of on-disk inodes.
  - Ex) inode tables : 3 ~ 7, inode size : 256 bytes
    - 4-KB block can hold 16 inodes.
    - The file system contains 80 inodes. (maximum number of files)



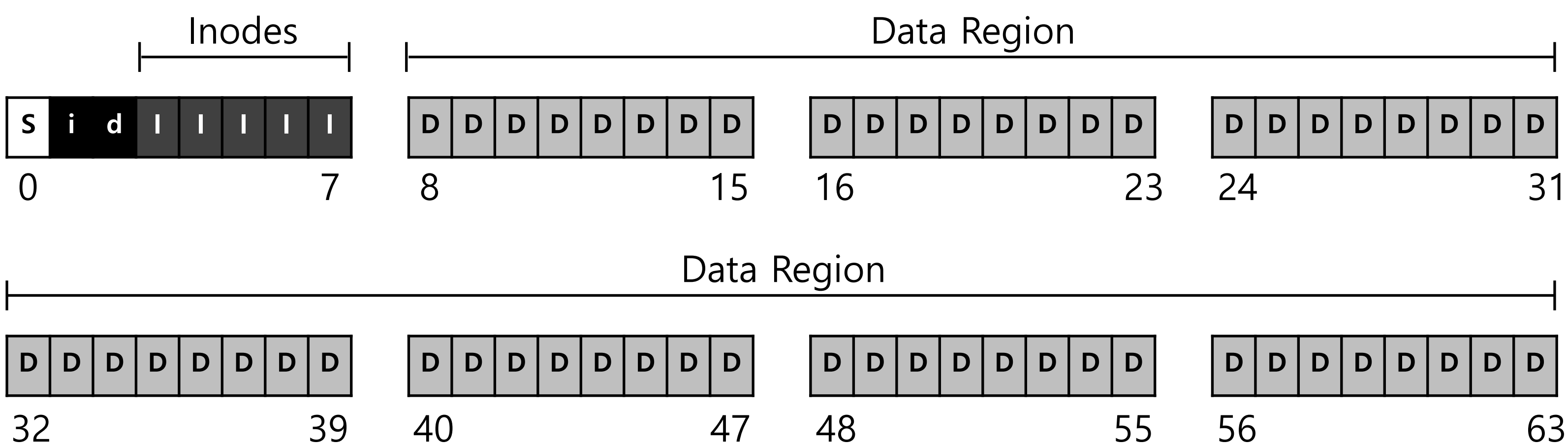
allocation structures

- This is to track whether inodes or data blocks are free or allocated.
- Use bitmap, each bit indicates free(0) or in-use(1)
  - data bitmap (d) : for data region
  - inode bitmap (i) : for inode table



super block

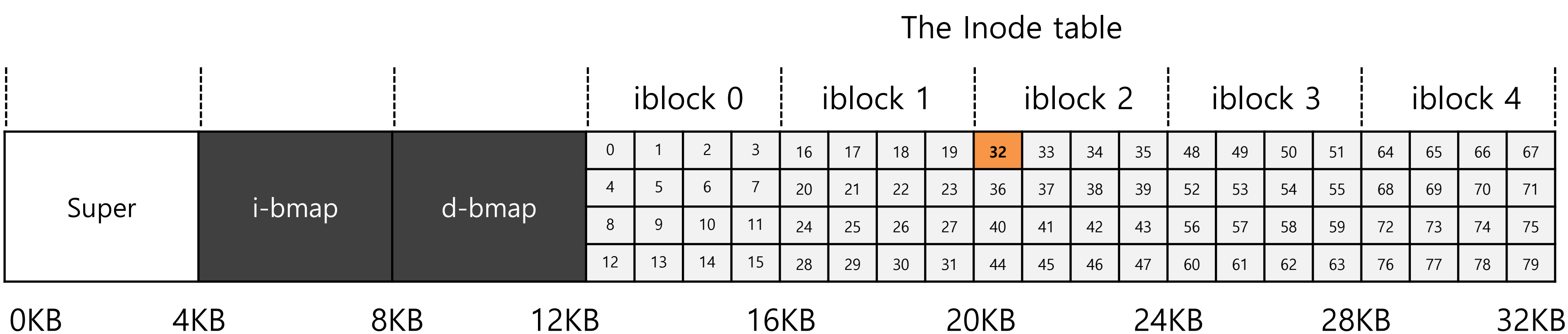
- Super block contains this information for particular file system
  - Ex) The number of inodes, begin location of inode table. etc



- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

File Organization: The inode

- Each inode is referred to by inode number.
  - by inode number, file system calculates where the inode is on the disk.
  - Ex) inode number: 32
    - Calculate the offset into the inode region (32 x sizeof(inode) (256 bytes) = 8192
    - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB



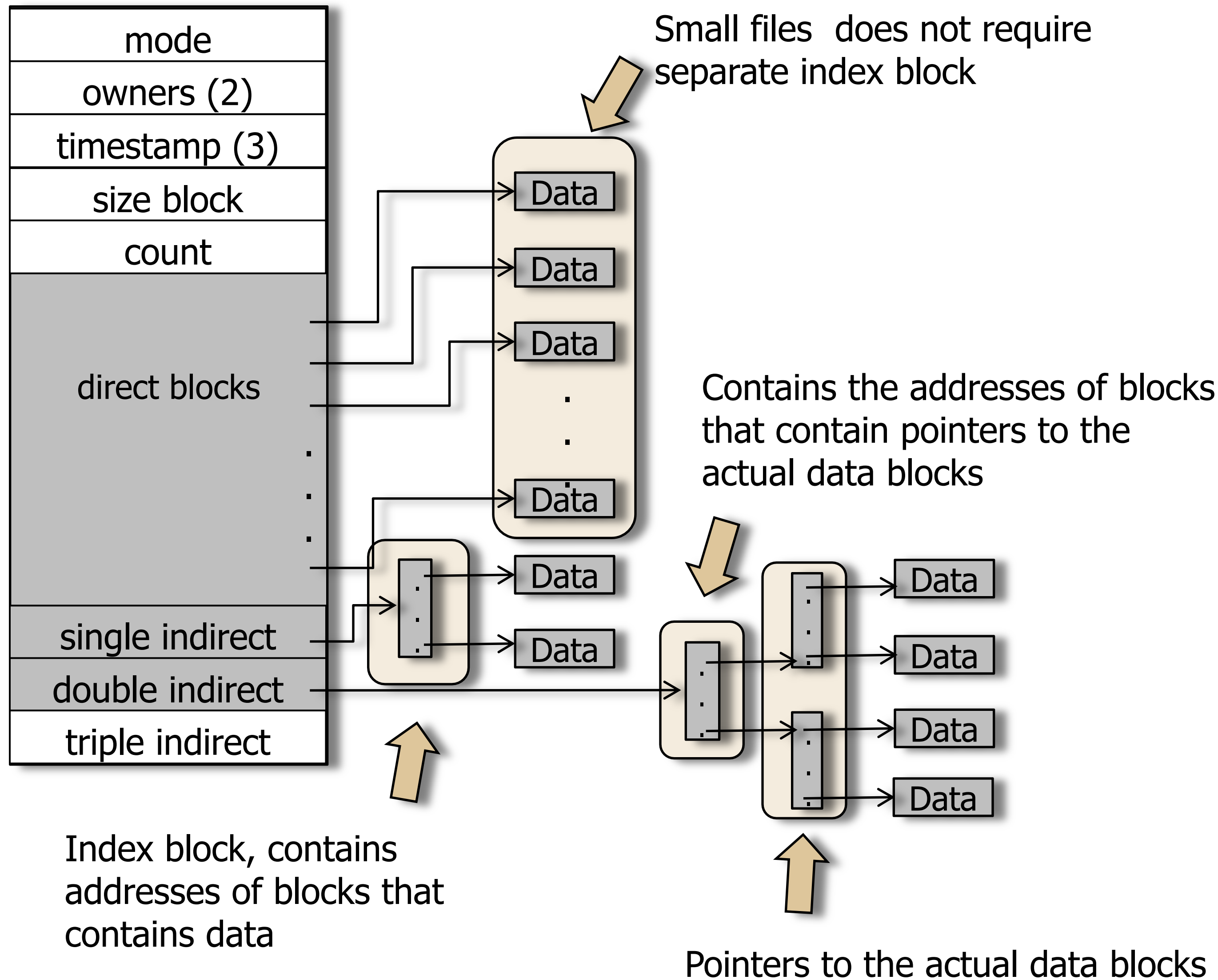
File Organization: The inode

- What's inside an inode?

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

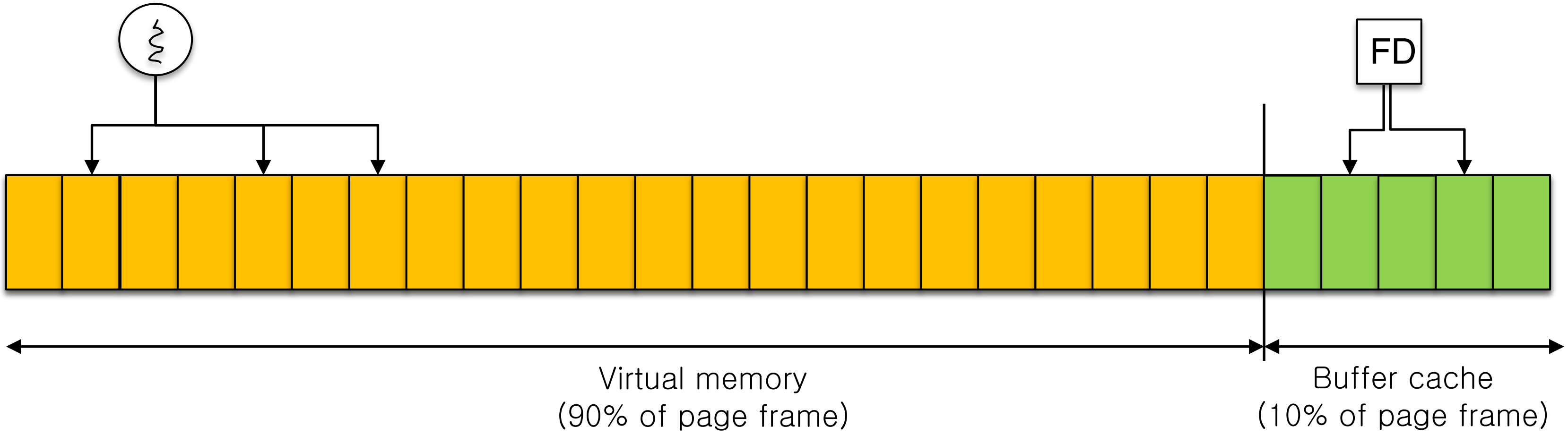
Simplified Ext2 Inode

File Structure: Indexed Allocation



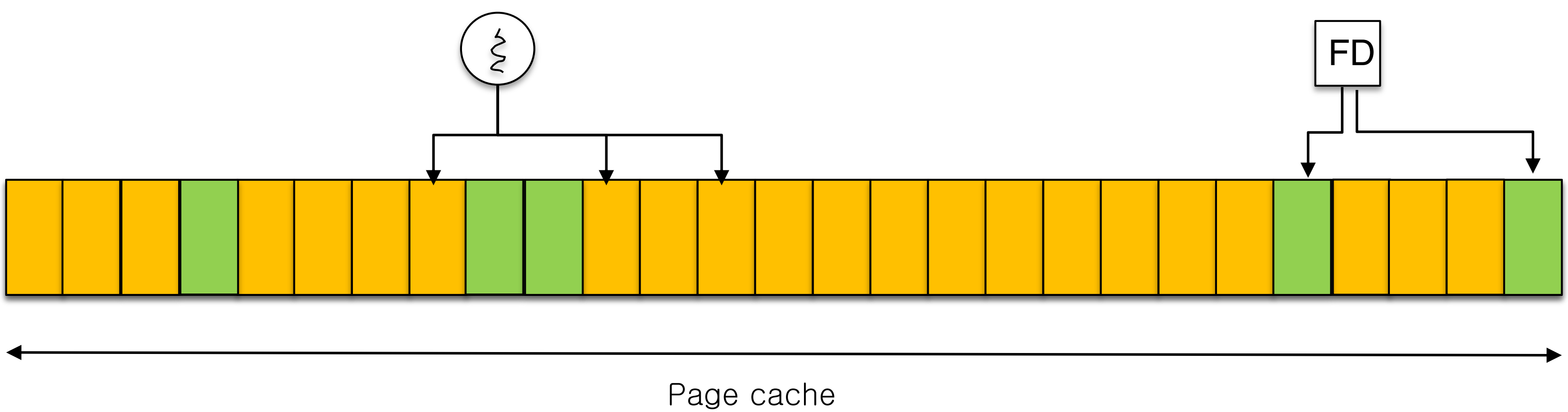
Caching and Buffering

- Reading and writing can very IO intensive.
  - File open: two IO for each directory inode and one read for the data inode.
- Buffer Cache
  - cache the disk blocks to reduce the IO.
  - LRU replacement
  - Static partitioning: 10% of DRAM, inefficient usage



Caching and Buffering

- Page Cache
  - Merge virtual memory and buffer cache
  - A physical page frame can host either a page in the process address space or a file block.
    - Process uses page table to map a virtual page to a page frame.
    - A file IO uses "`address_space`"(Linux) to map a file block to a physical page frame.
  - Dynamic partitioning



Summary

- Requirements for building filesystem
  - File information: inode
  - File structure: indexed file
  - Caching and buffering
- All are flexible.