Operating Systems

Lecture 14: File Systems Part II: API, A Simple FS

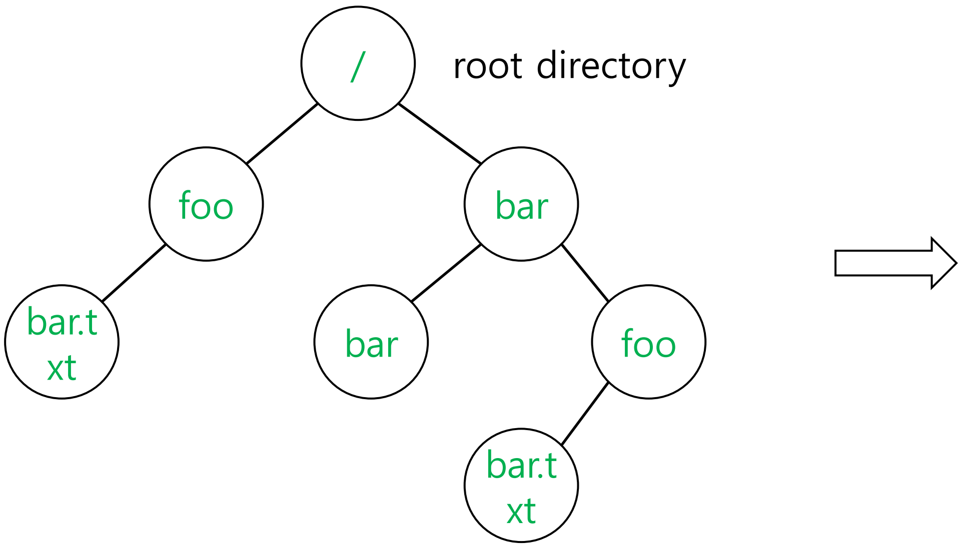
Hong Xu

https://github.com/henryhxu/CSCI3150

Files and Directories: API

Abstractions

- File
  - A file is data with some properties (name, type, permissions, etc.)
  - inode, “index node”, a data structure that describes a FS object
  - Each file has a low-level name as an ‘inode number’
- Directory
  - A file; A list of <user-readable filename, low-level name> pairs



An Example Directory Tree

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

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

File System Interface: Creating a file

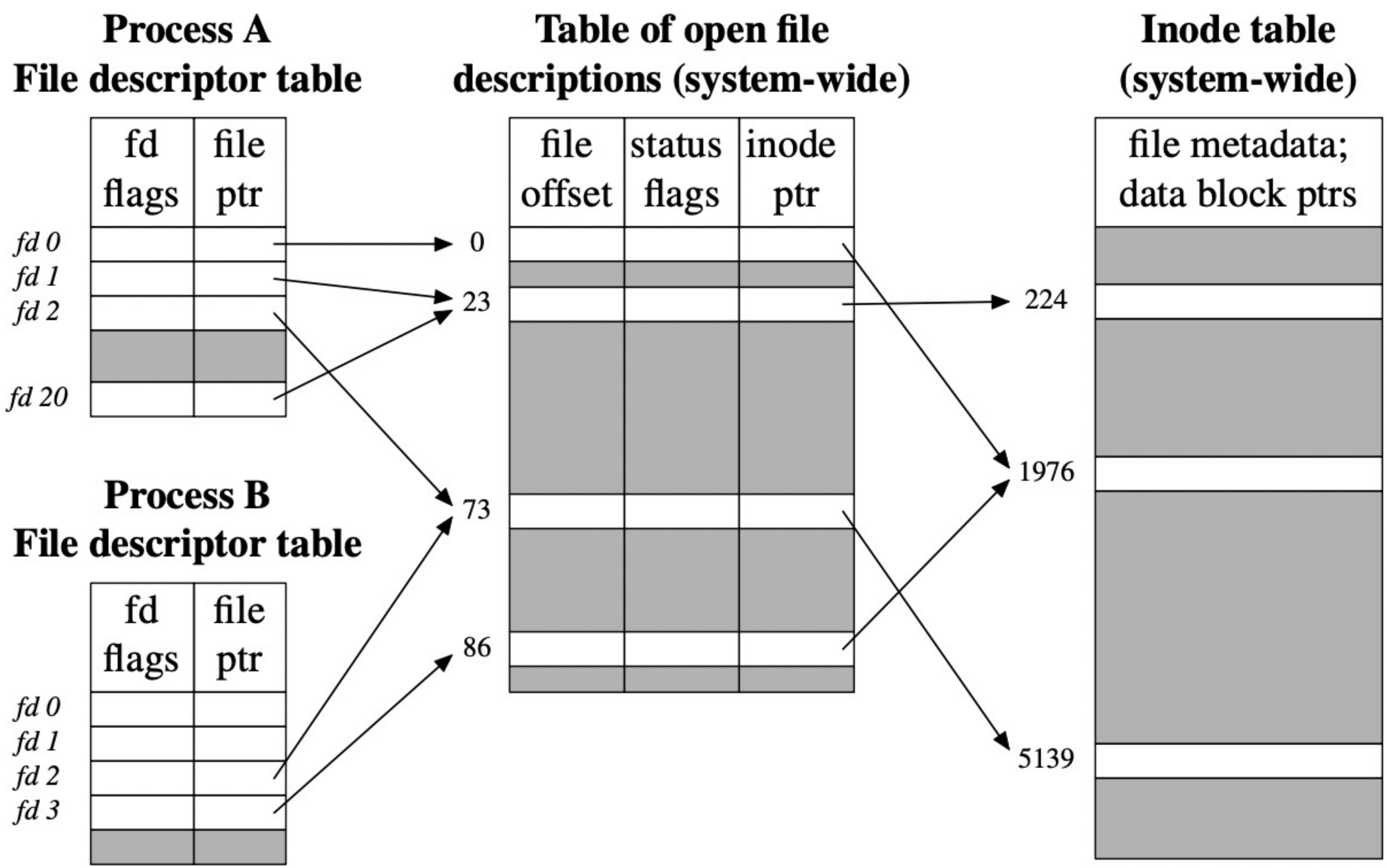
- Use `open` system call with `O_CREAT` flag

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

  - `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 a file descriptor
  - fd is an integer, is used to access files.
  - Ex) `read (file descriptor)`
  - File descriptor table

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

Aside: A Few Tables for Files in Unix/Linux

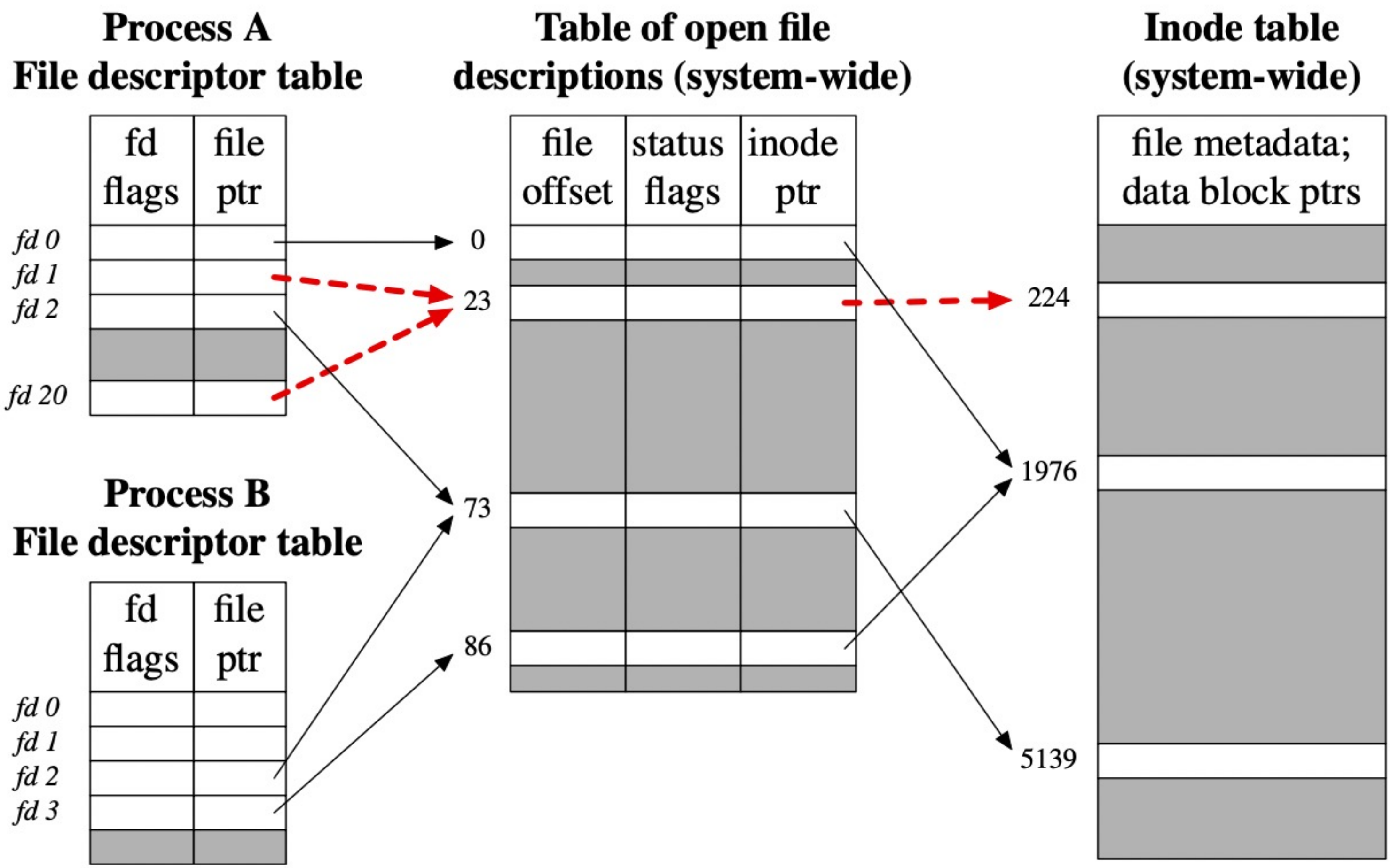


Aside: A Few Tables for Files in Unix/Linux

- File descriptor table
  - Per-process, one entry for each fd opened by this process
  - Flags controlling the operations of a fd
- Open file table
  - Global, system-wide, one entry for each file opened by "open"
  - File offset, access mode, etc.
  - Terms: Open file table entry, open file description (OFD) (POSIX)
- inode table
  - System-wide, one entry per inode in the fs
  - File type (regular file, socket, etc.)
  - Permissions, properties, etc.

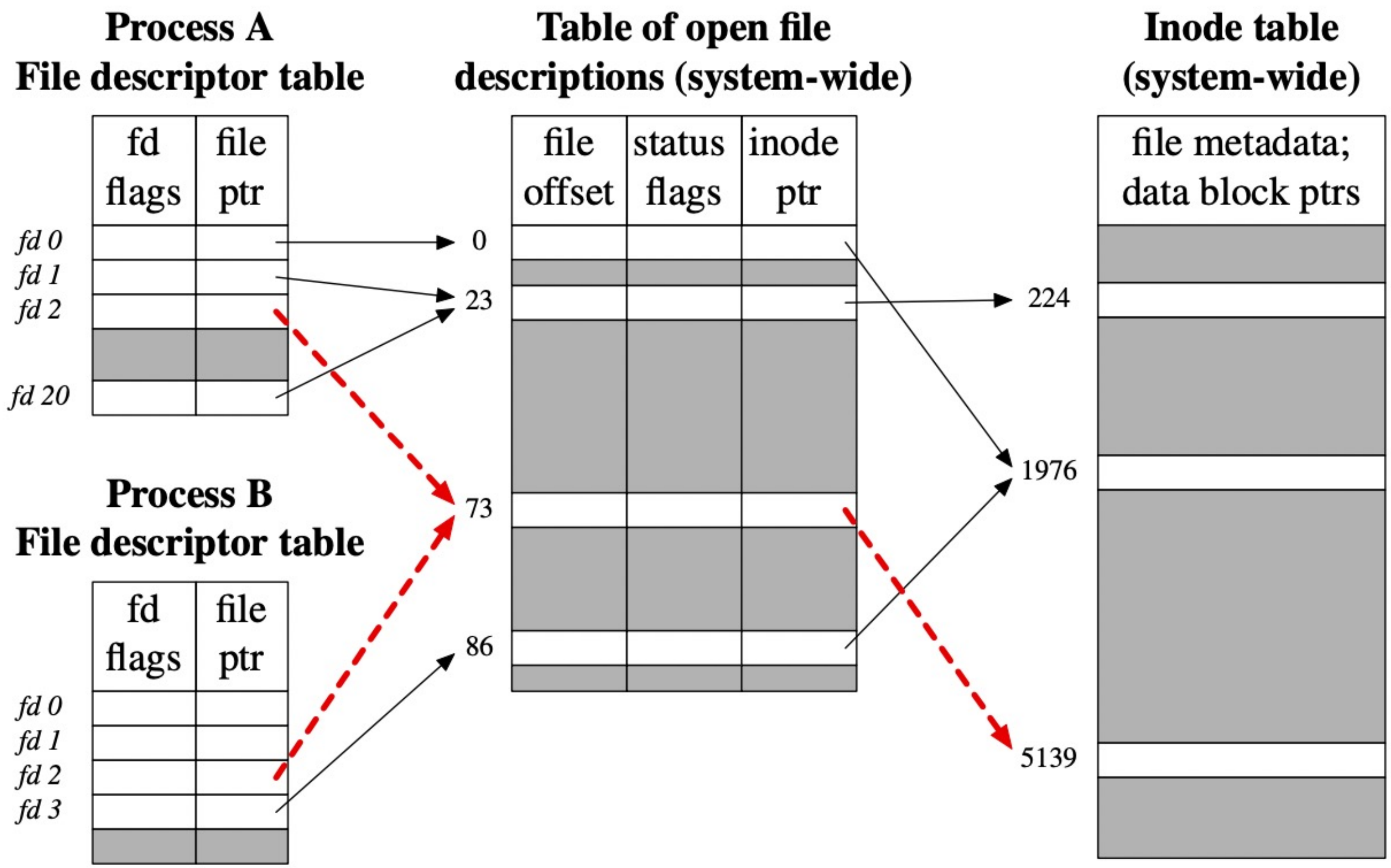
Duplicated File Descriptors (intra-process)

- A process may have multiple fds referring to the same OFD
  - Achieved with `dup()` or `dup2()`



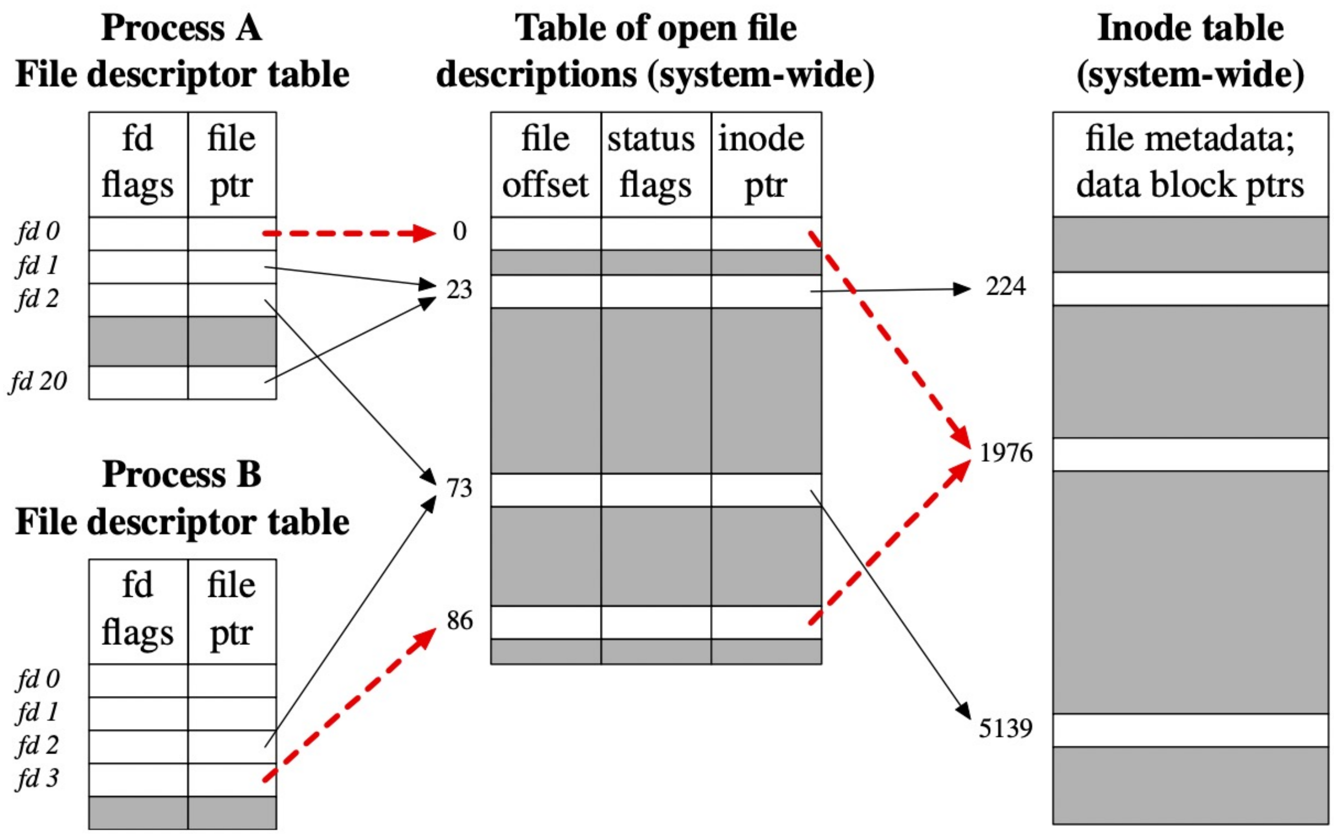
Duplicated File Descriptors (inter-process)

- Two processes may have fds referring to the same OFD
  - As a result of `fork()`



Duplicated OFDs

- Two processes may have fds referring to distinct OFDs, or open file table entries, that refer to the same inode
  - They independently opened the same file



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>
```

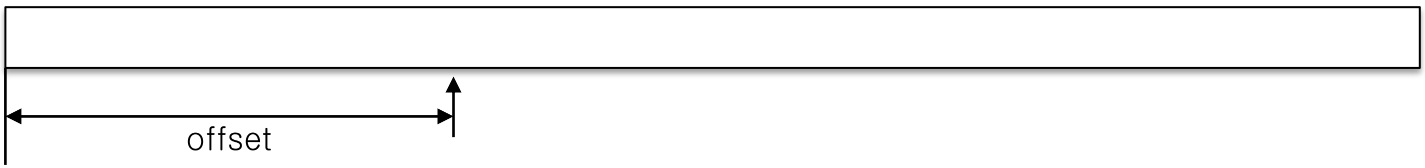
- The result of `strace` to figure out what `cat` is doing

```
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
read(3, "", 4096)                 = 0
..
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.

Data structures

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

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

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	–

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);
```

  - If a file is created, it needs to be durably a part of the directory.
    - Above code requires fsync() to directory also.

More API calls

- More about the file system API will be covered in the tutorials

A Simple File System

Overview

- We will study a very simple file system (vsfs)
  - Basic on-disk structures, access methods, and various policies of fs
- 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?

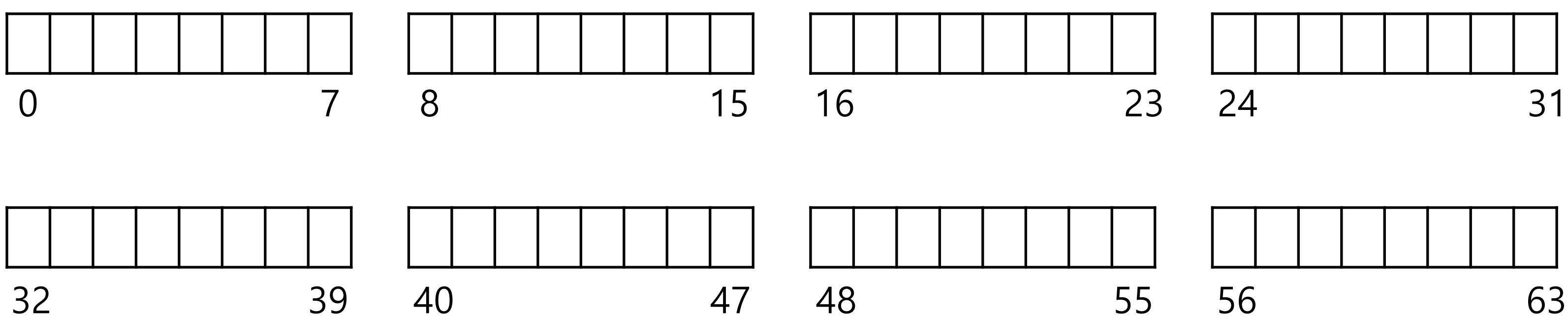
File System Implementation

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

Overall Organization

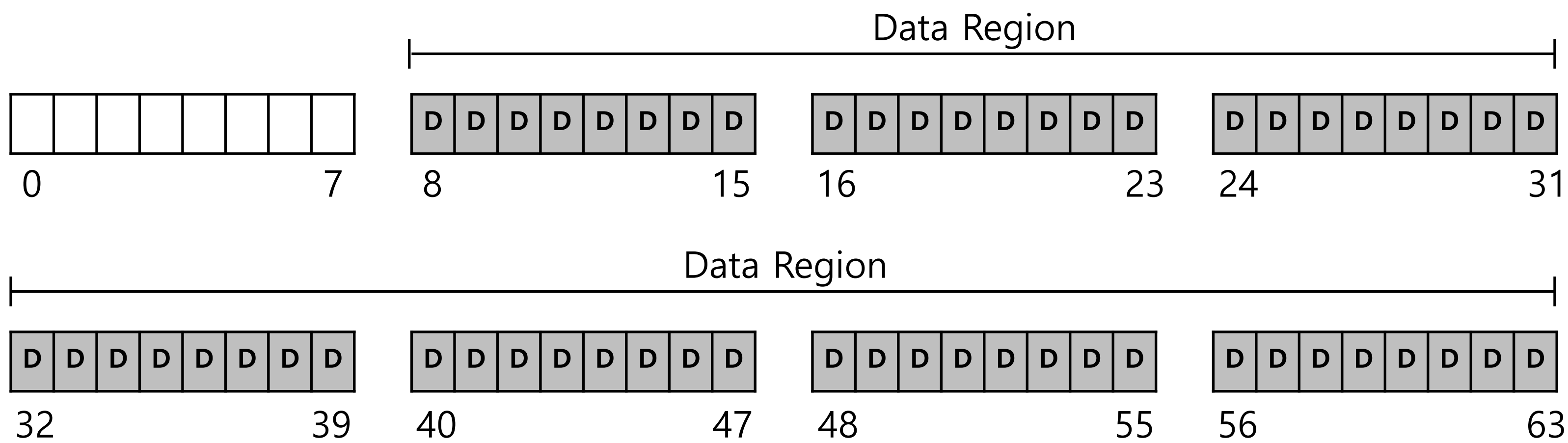
- Let's develop the overall organization of the file system data structure.

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



Data Region in a FS

- Reserve a data region to store user data

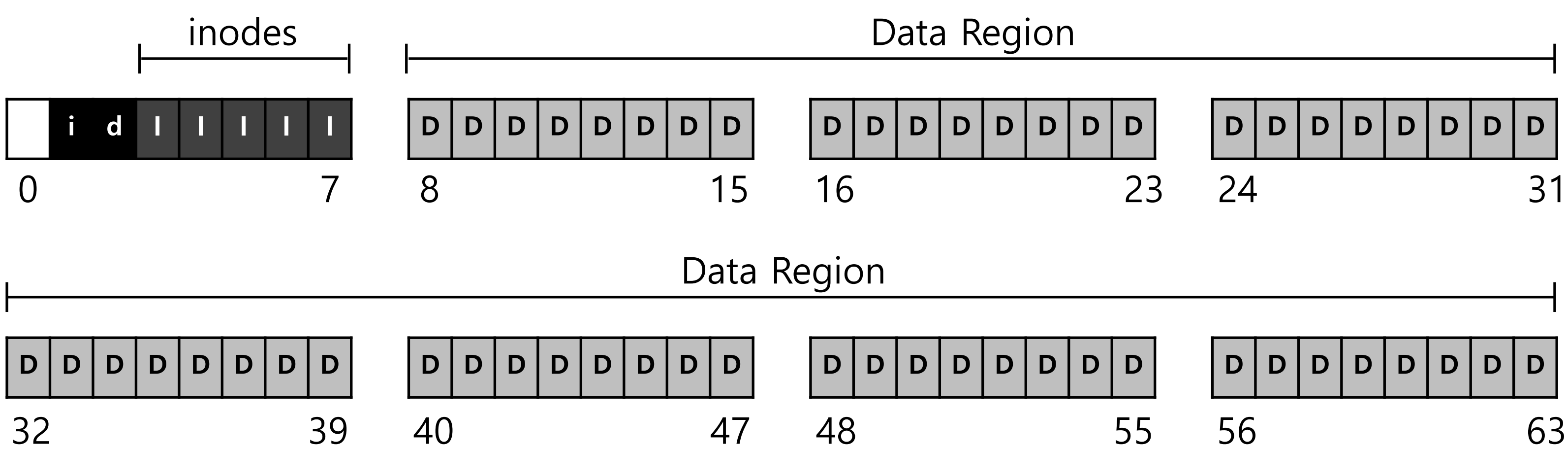


- The FS has to track which data blocks comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

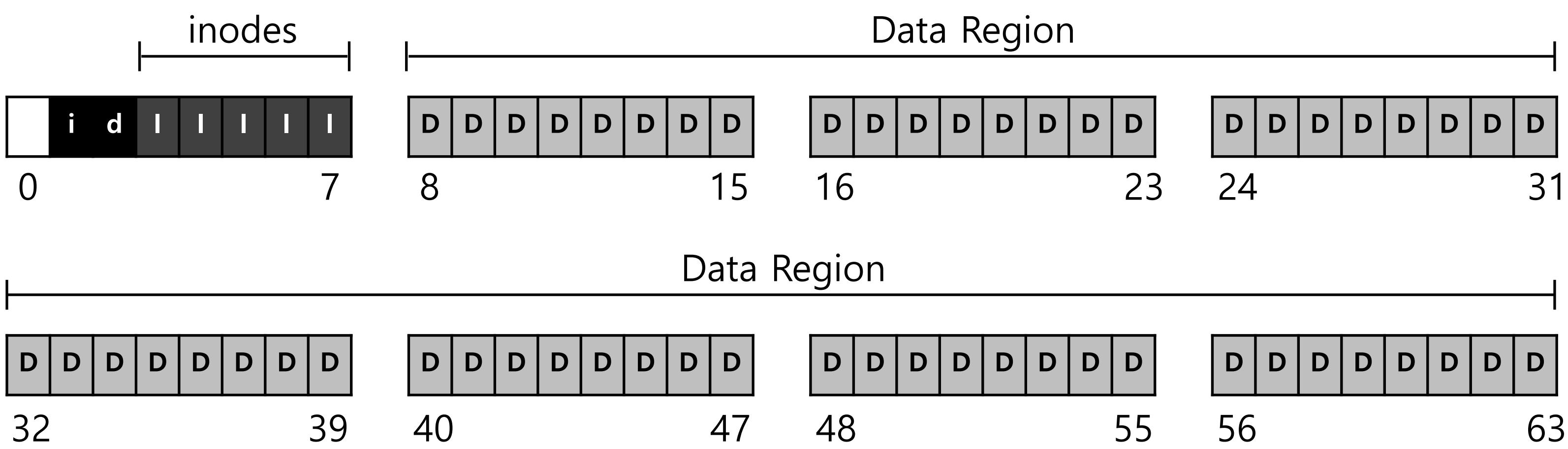
inode Table

- 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)



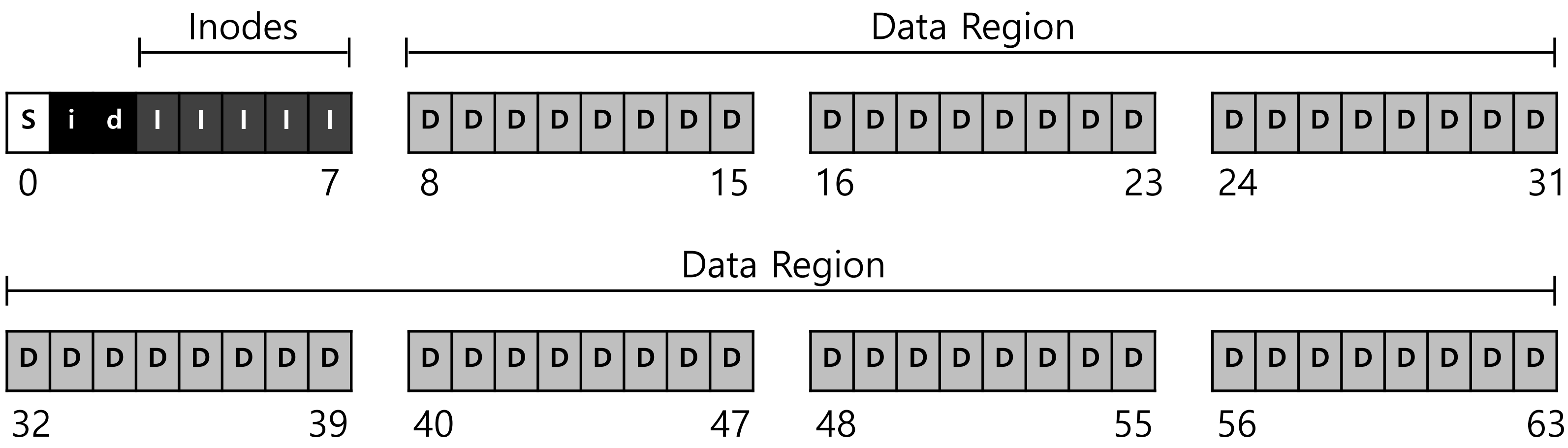
Allocation Structures

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



Super Blocks

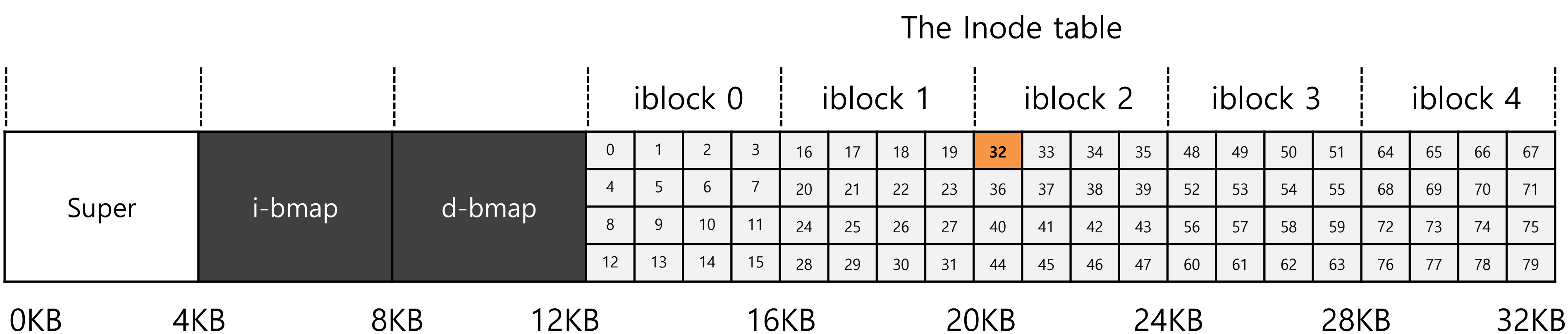
- Super block contains meta information for a file system
  - Ex) The number of inodes, begin location of inode table. etc



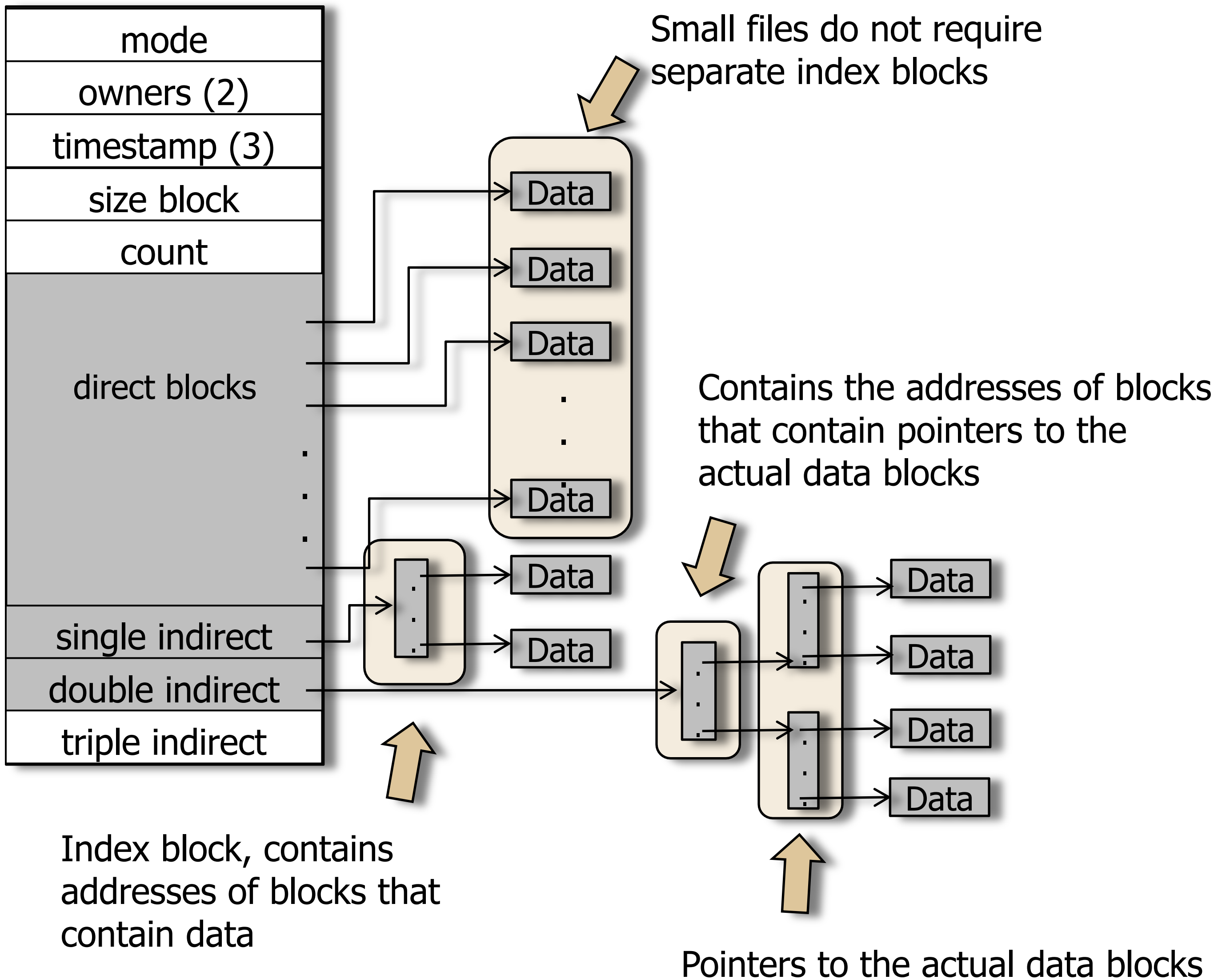
- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

File Organization: The inode

- Each inode is referred to by an inode number.
  - by inode number, the FS 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 Structure: Indexed Allocation



Directory Structure

VSFS

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

EXT4

	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

File Read

Read "/foo/bar"

	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)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

File Creation

	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							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					

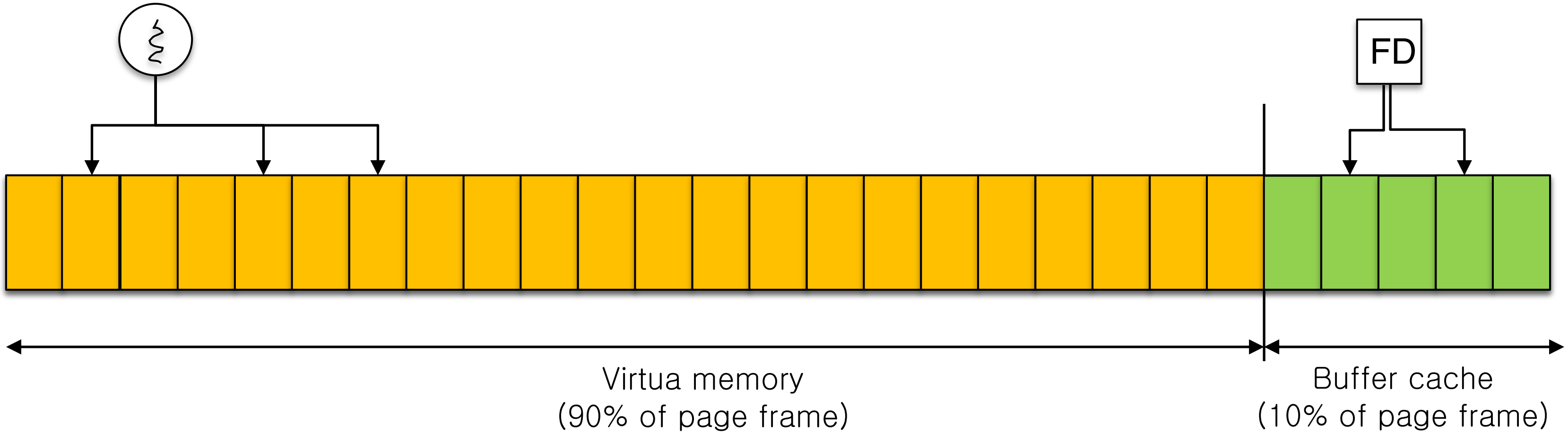
...

File Creation (Cont.)

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

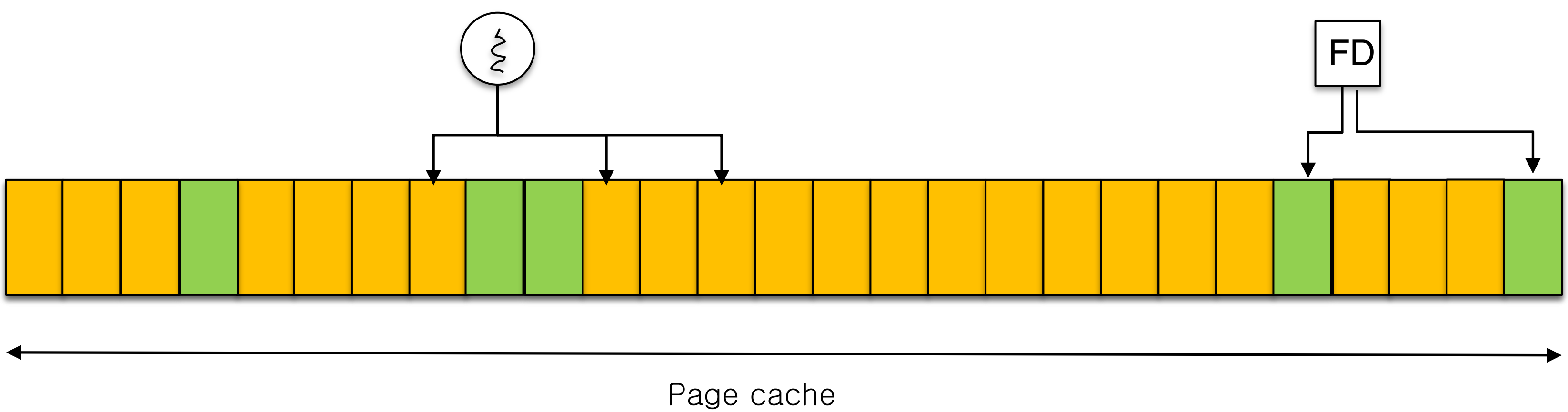
Caching and Buffering

- Reading and writing are very IO-intensive.
  - File open: two IO for each directory component and one read for the data.
- 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
  - Directory (name→inode-number): array of <inode #, name>'s
  - Free block information: Bitmap
- All are flexible.

How to Make it Faster?

Problem of Unix operating system

- Unix file system treated the disk as a **random-access memory.**

- Example of random-access blocks with four files.
  - Data blocks for each file can accessed by going back and forth the disk,

A1	A2	B1	B2	C1	C2	D1	D2

  - File b and d is deleted.

A1	A2			C1	C2		

  - File E is created with free blocks. (**spread across** the block)

A1	A2	E1	E2	C1	C2	E3	E4

FFS: Disk Awareness is the solution

- FFS is **Fast File System** designed by a group at Berkeley.
- The key idea of FFS is that file system structures and allocation polices need to be “disk aware” to improve performance.
  - Keep same API with file system. (`open()`, `read()`, `write()`, etc)
  - Changing the internal implementation.

Cylinder Groups

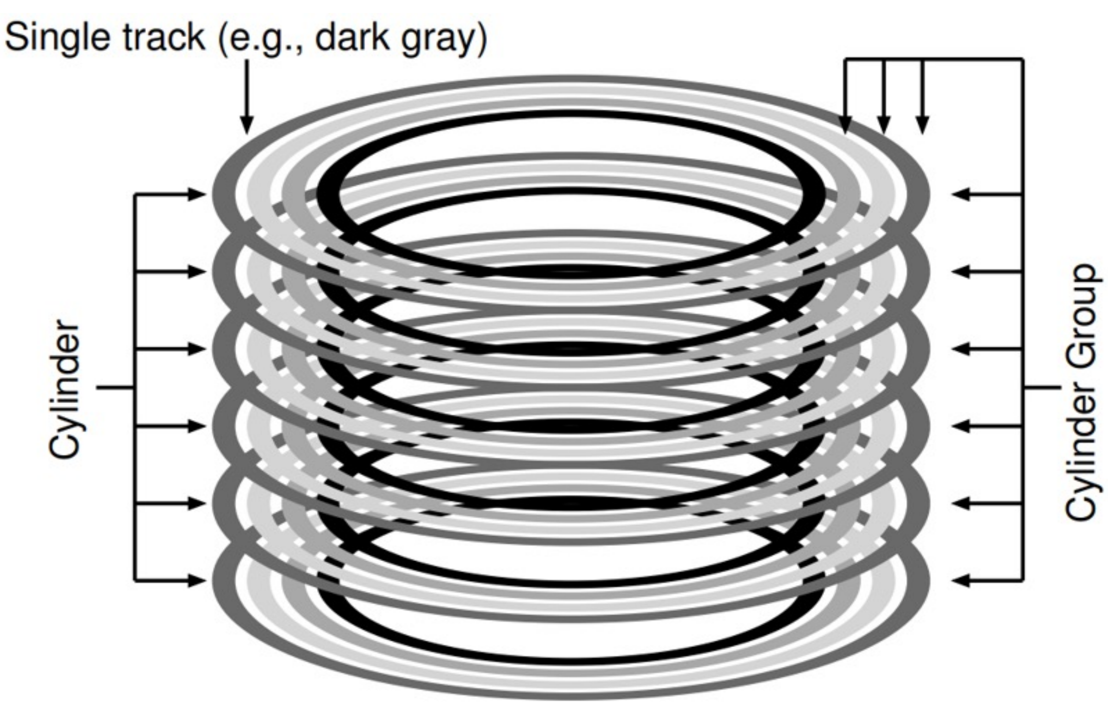
- FFS divides the disk into a bunch of groups, **Cylinder Groups**
  - Modern file system call cylinder group as block group.
- These groups are used to improve seek performance.
  - By placing two files within the same group.
  - Accessing one after the other **will not be long seeks** across the disk.
  - FFS needs to allocate the files and directories within each of these groups.

S	ib	db	Inodes	Data

- Data structure for each cylinder group.
  - A copy of the super block for reliability reason.
  - inode bitmap and data bitmap to track free inode and data block.
  - inodes and data block

Cylinder Group (Cont.)

- **Cylinder**: Tracks at same distance from center of drive across different surfaces.
  - All tracks with same color
- **Cylinder Group**: Set of N consecutive cylinders
  - if N=3, first group does not include black track



How To Allocate Files and Directories?

- Policy is “**keep related stuff together**”
- The placement of directories
  - Find the cylinder group with a low number of allocated directories and a high number of free inodes.
  - Put the directory data and inode in that group.
- The placement of files.
  - Allocate data blocks of a file in the same group as its inode
  - It places all files in the same group as their directory

Summary

- The introduction of fast file system (FFS)
  - It makes file system fast, considering characteristic of disk.
- Many file systems take cues from FFS.
  - ex) ext4, ext3, ext4