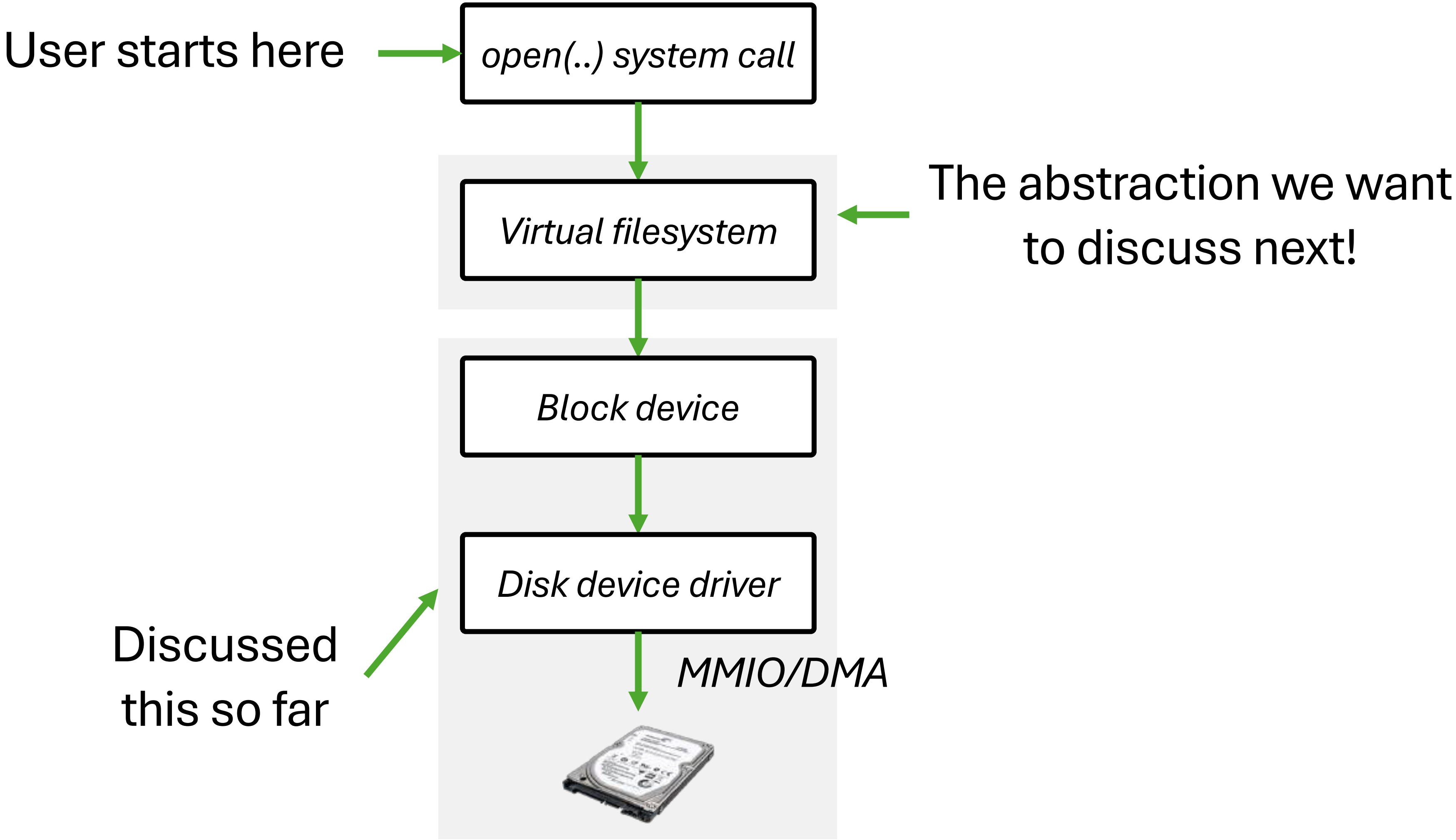
CSE 330: Operating Systems

Adil Ahmad

Lecture #18: File system overview and internals

The (virtual) file system recap

Disk interactions starting from a system call



User starts here

open(..) system call

Virtual filesystem

The abstraction we want to discuss next!

Block device

Disk device driver

MMIO/DMA

Discussed this so far

The (virtual) filesystem abstraction

- High-level intuitive view of how we look at data stored on disks
- Not just disk-related; UNIX philosophy → "everything is a file"

Can anyone tell me of other ways in which you have used files?

- High-level intuitive view of how we look at data stored on disks
- Not just disk-related; UNIX philosophy → "everything is a file"
  - /dev/memalloc → virtual file to communicate with modules
  - CPU features can be enabled or disabled using files.
    - E.g., entire CPUs can be disabled as follows:

      ```
      echo 0 | sudo tee /sys/devices/system/cpu/cpu1/online
      ```
  - Perform console R/W → write(1, "hello", 5)
    - 1 → file descriptor for console (terminal) output

Let’s get back to (disk-related) file

- File → a set of blocks that the OS has combined and operates on together
    - Recall that the filesystem is composed of the the block layer
- Files are given identifiers (e.g., “hello.txt” is a human-readable version) so programs/users can distinguish between them
- File system (FS) → an intricate hierarchical collection of files built using the blocks in your storage disk

Different “names” for a file

- Three different names typically
  - ✓ **inode** (low-level names)
    - ➢ Internal name (number) given to a file by the OS
  - ✓ **path** (human readable)
    - ➢ The version that we see when we open the file browser (e.g., Windows explorer or MacOS finder)
  - ✓ **file descriptor** (runtime state)
    - ➢ Represents the runtime status of a certain file

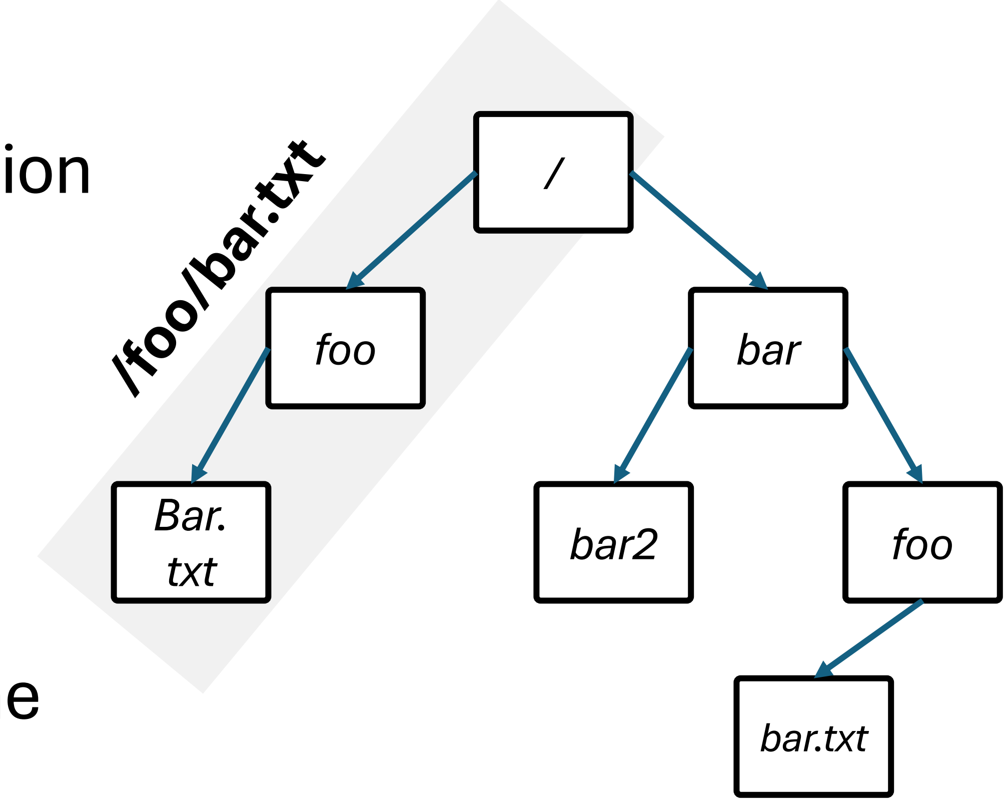
The inode (OS-level representation)

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- **File names can be the same, why can't inodes?**
    - Something must be unique for the OS to track

```
PROMPT>: stat test.dat
        File: 'test.dat'
        Size: 5
        Blocks: 8
        IO Block: 4096   regular file
        Device: 803h/2051d
        Inode: 119341128
        Links: 1
        Access: (0664/-rw-rw-r--)
        Uid: ( 1001/    yue)
        Gid: ( 1001/    yue)
        Context: unconfined_u:object_r:user_home_t:s0
        Access: 2015-12-17 04:12:47.935716294 -0500
        Modify: 2014-12-12 19:25:32.669625220 -0500
        Change: 2014-12-12 19:25:32.669625220 -0500
        Birth: -
```

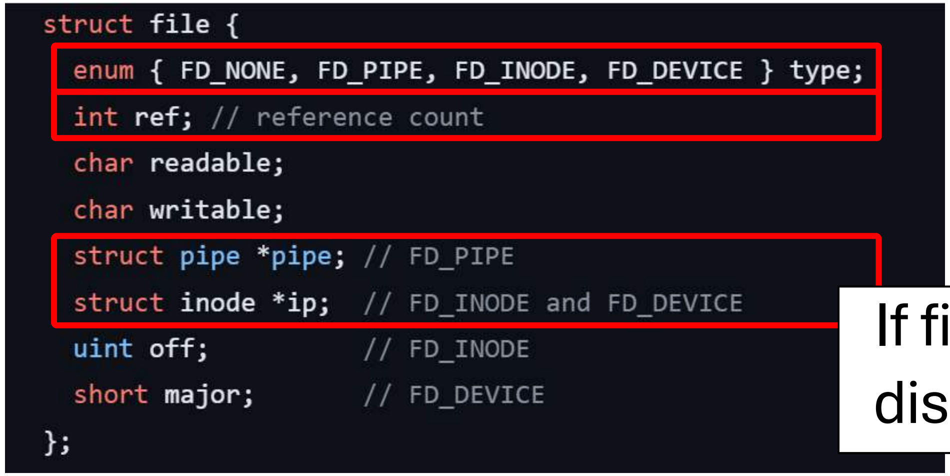
The path (human-readable)

- Human-readable interpretation of every inode
- Typically, represented as – <path-to-directory , filename>
- **Traversing** a tree – getting the final *inode* from a location
  - E.g., ls /foo/bar.txt
  - Gets the inode and prints details



The file descriptor

- “Everything is a file”
  - File descriptor tracks what each ‘file’ really does in the system



```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

If file belongs to the disk, it has an inode

Commonly-used file system interfaces

Creating a new file or opening an existing file

- UNIX system call: open()

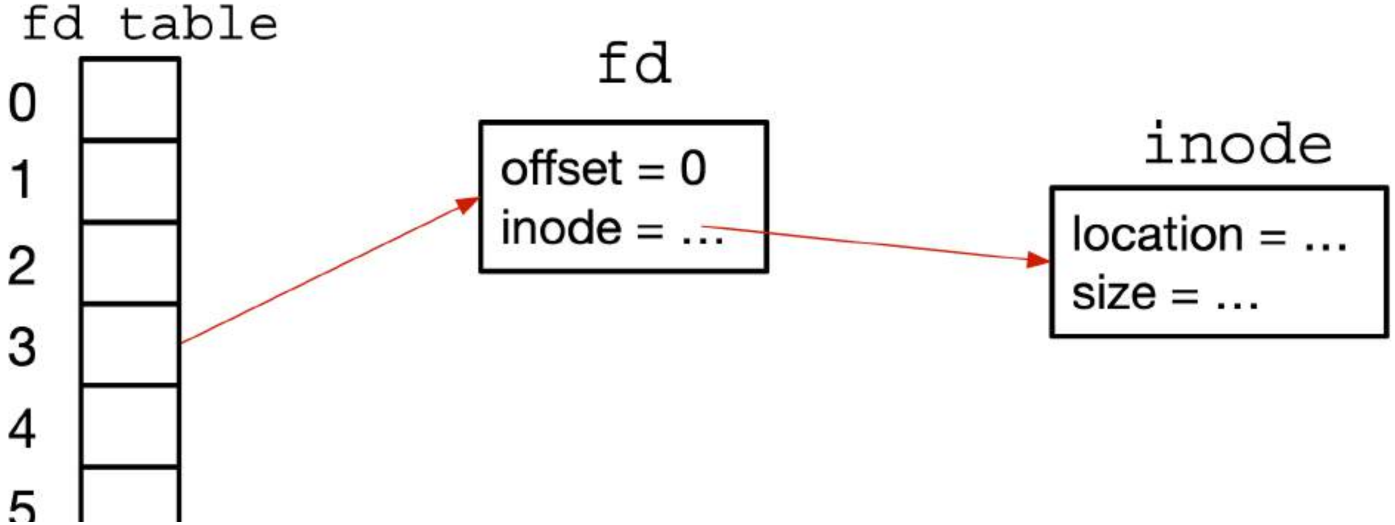
```
int fd = open(char *path, int flag, mode_t mode);
-OR-
int fd = open(char *path, int flag);
```

- open() returns a file descriptor (fd)
  - A fd is an integer
  - Private per process

- fd is a handle that gives caller the power to perform certain operations
  - You can think of a “fd” as a pointer to an object of the file

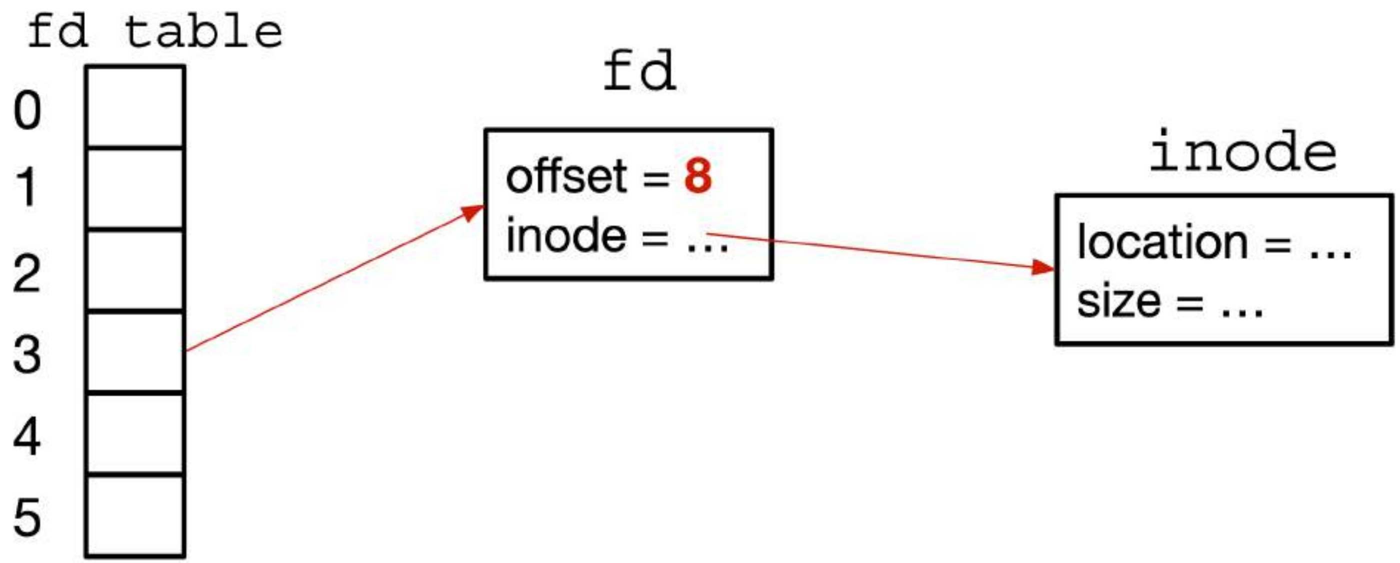
open() example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
```



open() example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
```



fd table

0
1
2
3
4
5

fd

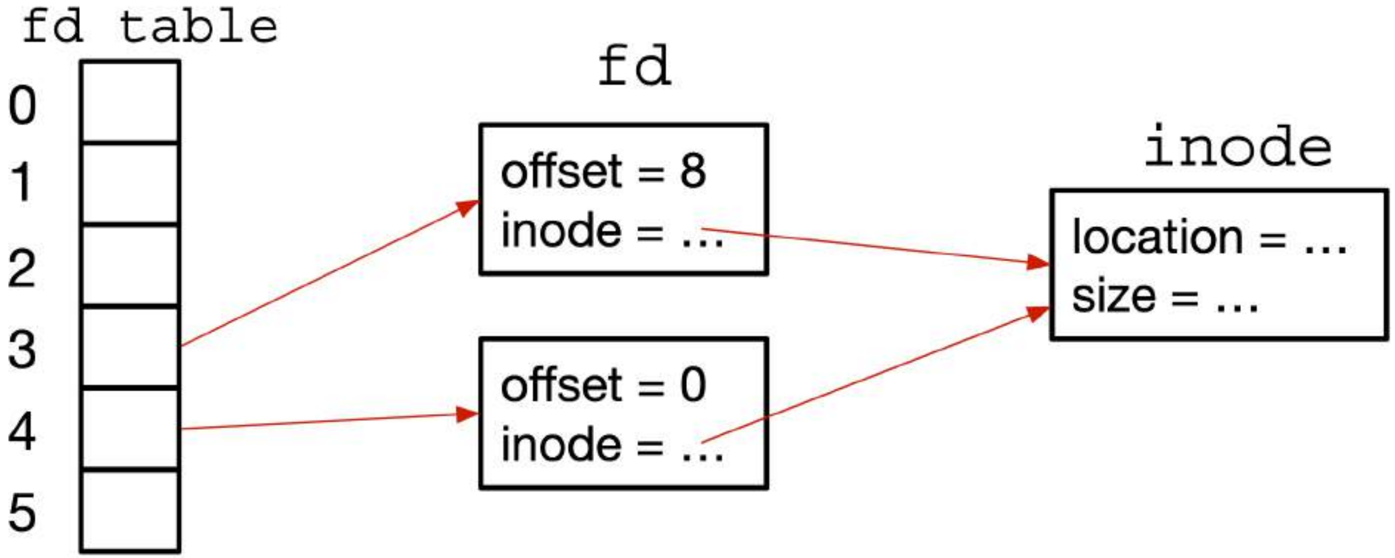
offset = 8
inode = …

inode

location = …
size = …

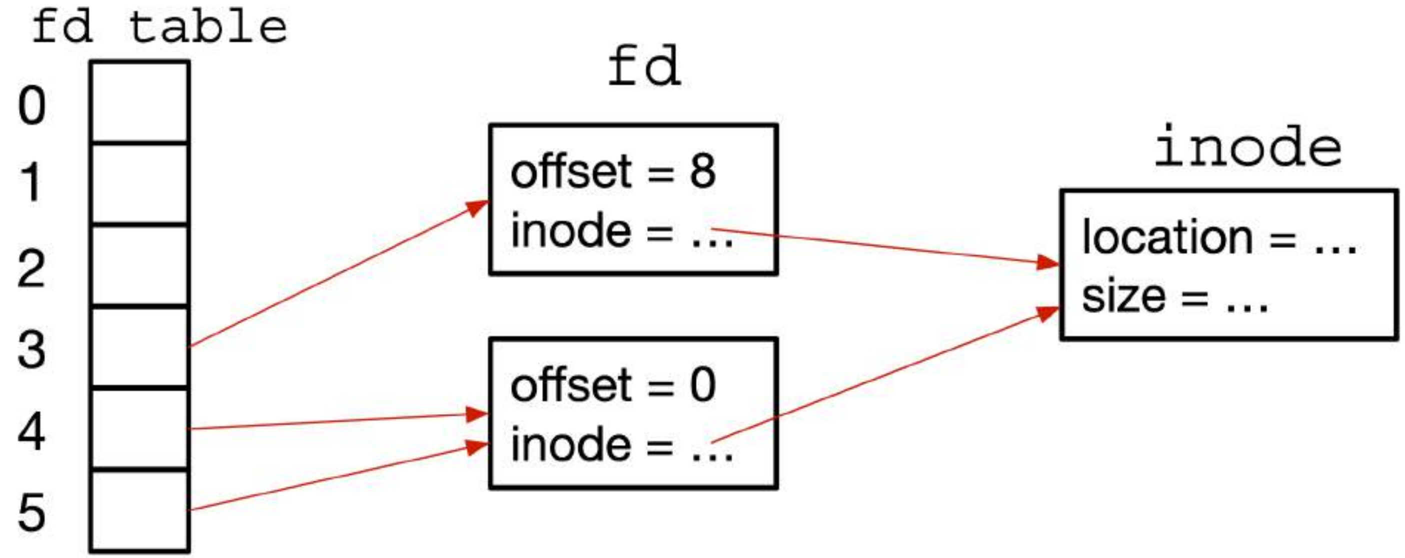
open() example

```
int fd1 = open(“file.txt”, O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open(“file.txt”, O_WRONLY); // return 4
```



`open()` example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open("file.txt", O_WRONLY); // return 4
int fd3 = dup(fd2);                   // return 5
```



fd table

0
1
2
3
4
5

fd

offset = 8
inode = …

offset = 0
inode = …

inode

location = …
size = …

Other common file system interfaces

- UNIX system call: read() and write()

```
int ret = read(int fd, char *buffer, size_t size);
int ret = write(int fd, char *buffer, size_t size);
```

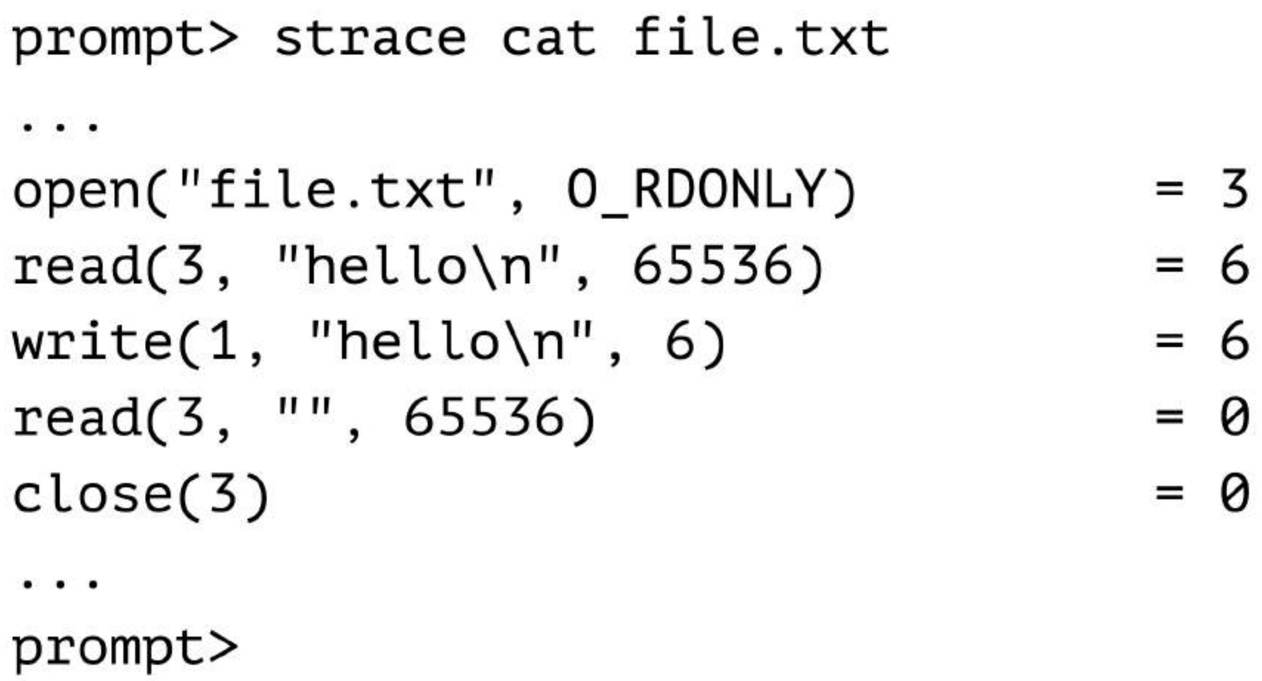
- **Can anyone tell me why they have the same arguments?**

Can anyone tell me what FS calls this command invokes?

- cat test.txt

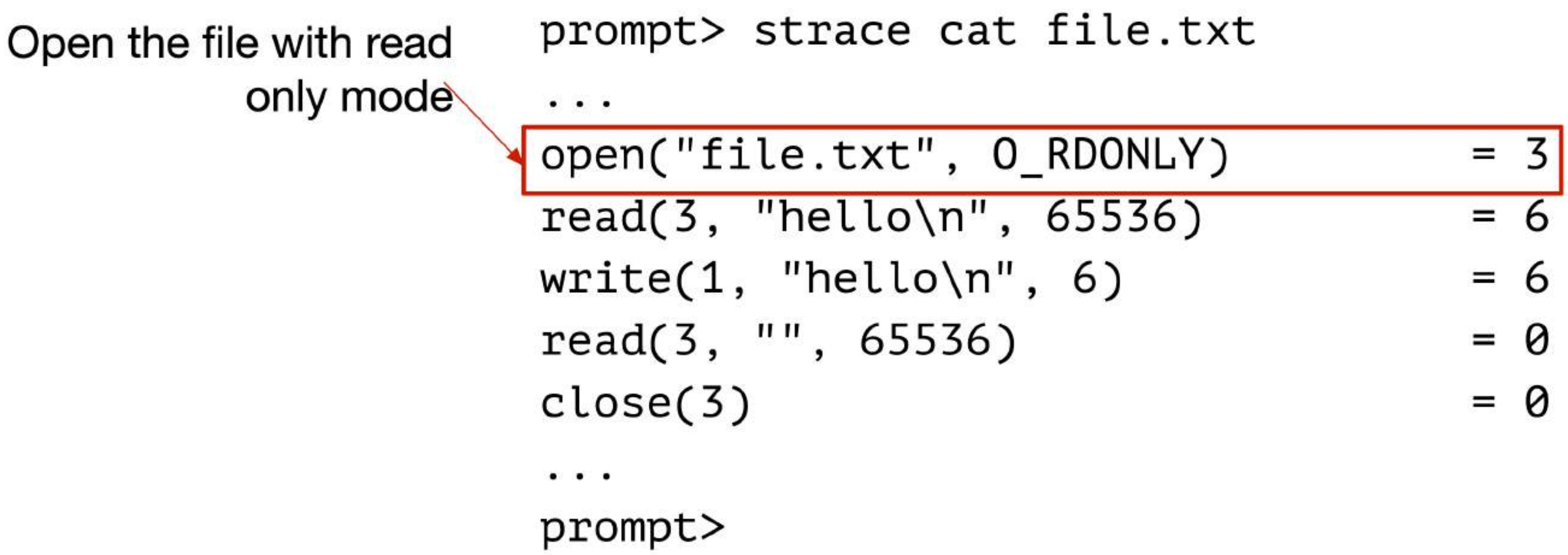
Can anyone tell me what FS calls this command invokes?

- cat test.txt



```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                = 3
read(3, "hello\n", 65536)                 = 6
write(1, "hello\n", 6)                    = 6
read(3, "", 65536)                        = 0
close(3)                                  = 0
...
prompt>
```

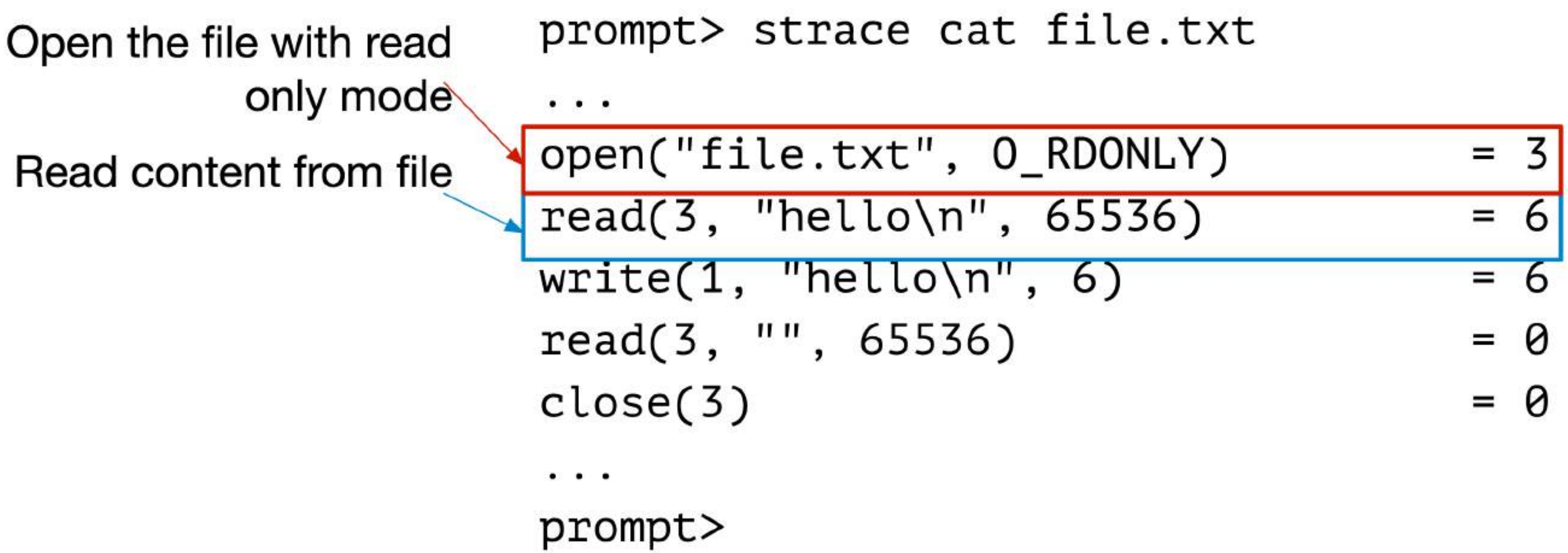
Can anyone tell me what FS calls this command invokes?



Open the file with read only mode

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                  = 3
read(3, "hello\n", 65536)                   = 6
write(1, "hello\n", 6)                      = 6
read(3, "", 65536)                          = 0
close(3)                                    = 0
...
prompt>
```

Can anyone tell me what FS calls this command invokes?

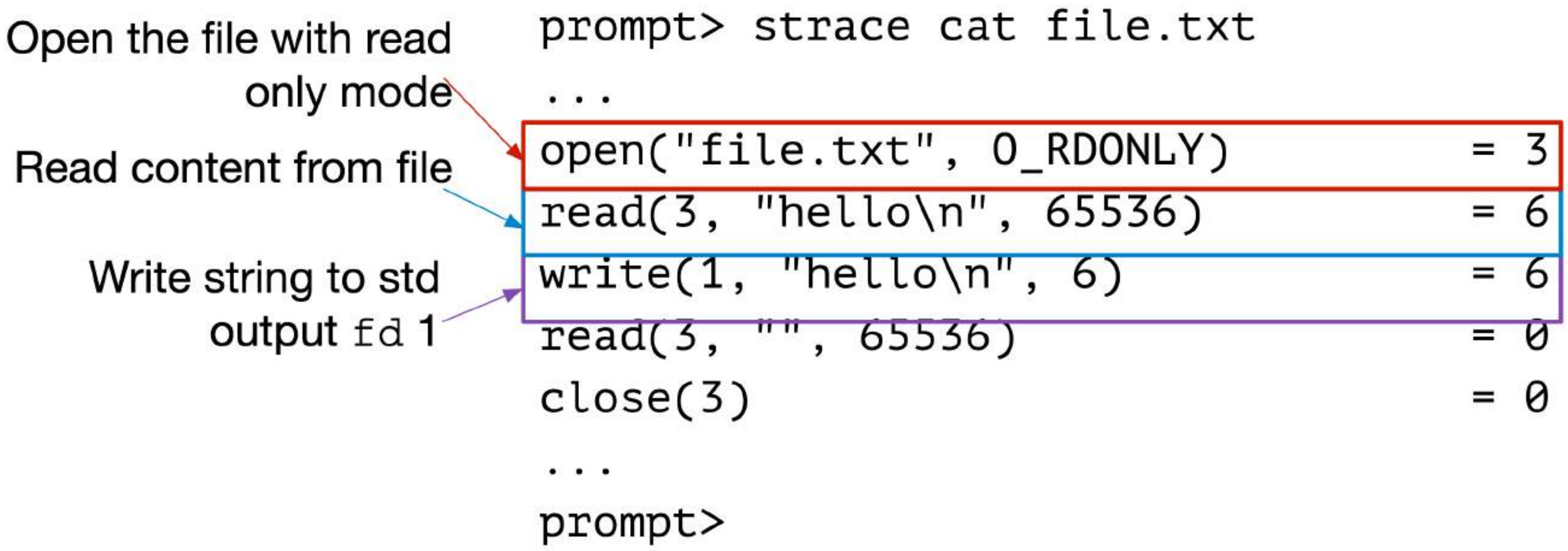


```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                 = 3
read(3, "hello\n", 65536)                  = 6
write(1, "hello\n", 6)                     = 6
read(3, "", 65536)                         = 0
close(3)                                   = 0
...
prompt>
```

Open the file with read only mode

Read content from file

Can anyone tell me what FS calls this command invokes?



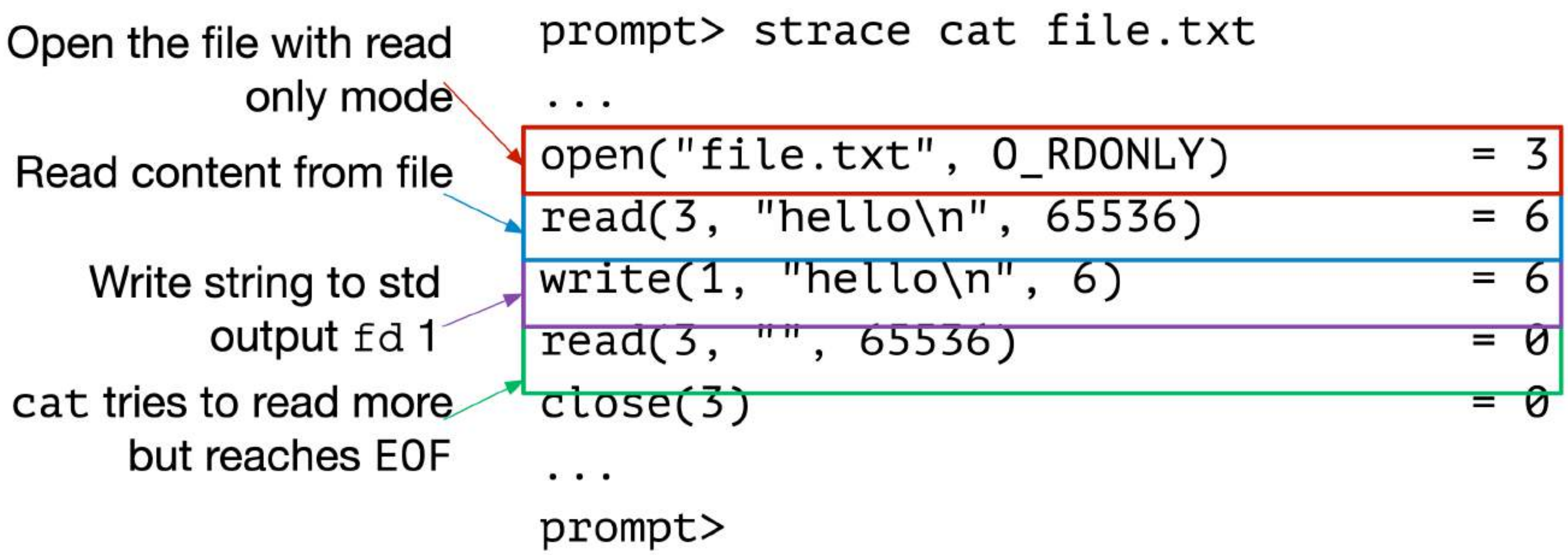
```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)              = 3
read(3, "hello\n", 65536)               = 6
write(1, "hello\n", 6)                  = 6
read(3, "", 65536)                      = 0
close(3)                                = 0
...
prompt>
```

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

Can anyone tell me what FS calls this command invokes?



```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                = 3
read(3, "hello\n", 65536)                 = 6
write(1, "hello\n", 6)                    = 6
read(3, "", 65536)                        = 0
close(3)                                  = 0
...
prompt>
```

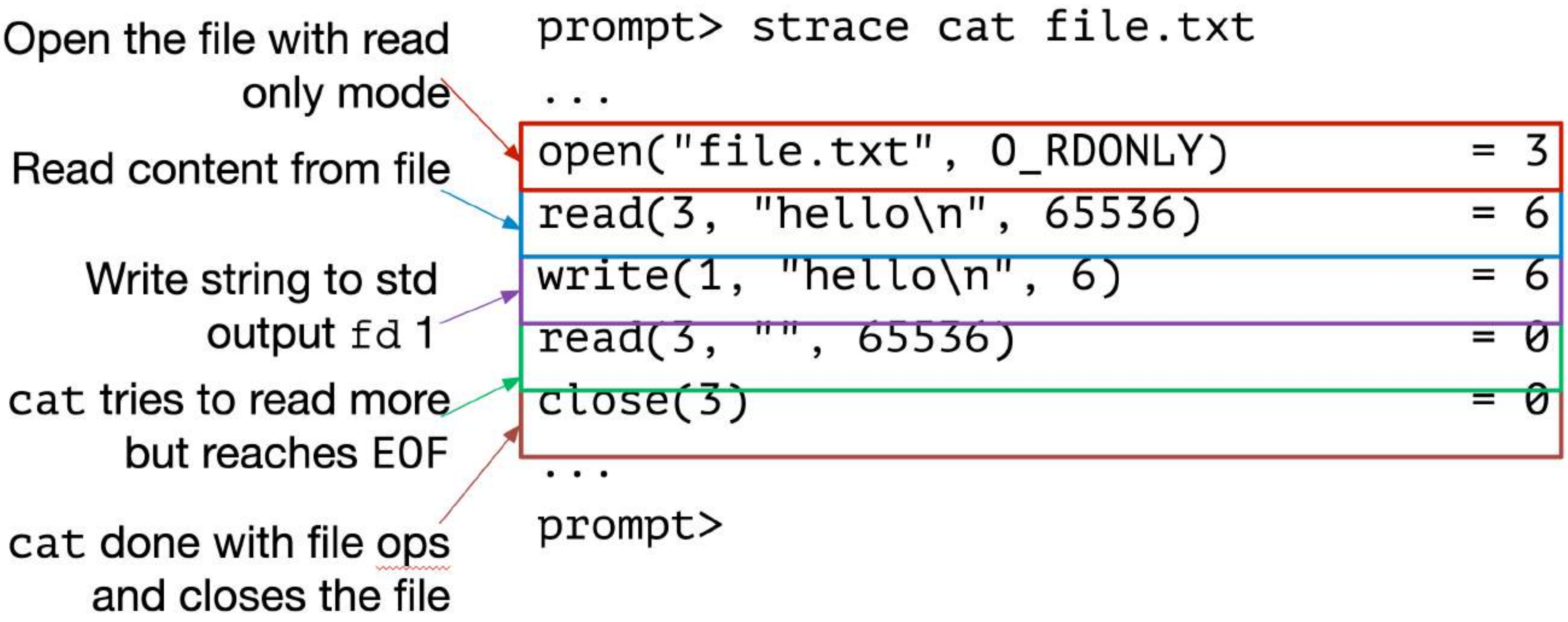
Open the file with read only mode

Read content from file

Write string to std output `fd` 1

cat tries to read more but reaches EOF

Can anyone tell me what FS calls this command invokes?



```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)                   = 3
read(3, "hello\n", 65536)                    = 6
write(1, "hello\n", 6)                       = 6
read(3, "", 65536)                           = 0
close(3)                                     = 0
...
prompt>
```

Open the file with read only mode

Read content from file

Write string to std output fd 1

cat tries to read more but reaches EOF

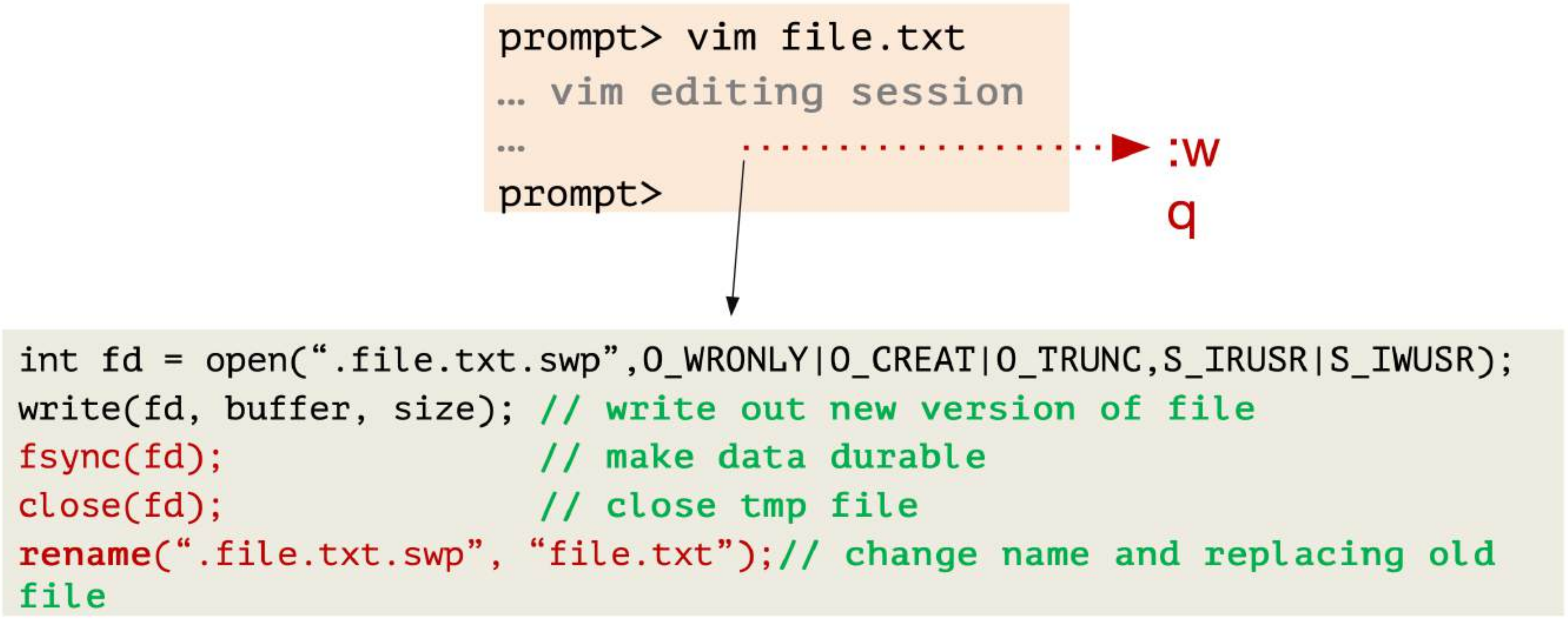
cat done with file ops and closes the file

Are writes to disk performed exactly when you call write()?

- No, that would be very slow since more writes could be needed later, and they can be batched together
  - ➢ Hence, your FS will buffer writes *in-memory*

- Sometimes though, you may want to force writes, e.g., if a later operation depends on disk write or to ensure persistence.
  - For such cases, FS provides the `fsync()` system call

Example of how “vim” uses the filesystem



```
prompt> vim file.txt
… vim editing session
…
prompt>
```

:w
q

```
int fd = open(“.file.txt.swp”,O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);               // make data durable
close(fd);               // close tmp file
rename(“.file.txt.swp”, “file.txt”);// change name and replacing old
file
```

Implementation of a filesystem

What are the questions we may need to answer to build an FS?

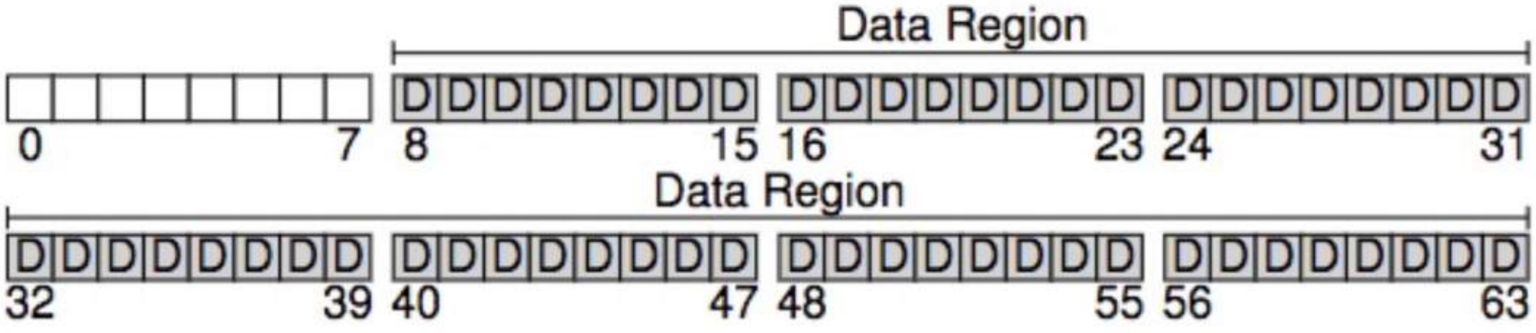
- Where to store the files in the disk?

- Where/how to store the information about existing files (e.g., inodes, path, blocks, etc.) in the disk?

- How to quickly find free locations in the disk for new files?

Storing file “data blocks” within the disk

- Initially, the entire disk is empty
- We will save some space to store information
- Use the rest to store our “data blocks” (figure below)



Data Region
D D D D D D D D D D D D D D D D D D D D D D D D
0
7
8
15
16
23
24
31
Data Region
D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
32
39
40
47
48
55
56
63

Now, we must answer:

- Where to store the files in the disk?

- Where/how to store the information about existing files (e.g., inodes, path, blocks, etc.) in the disk?

- How to quickly find free locations in the disk for new files?

An **inode** data structure

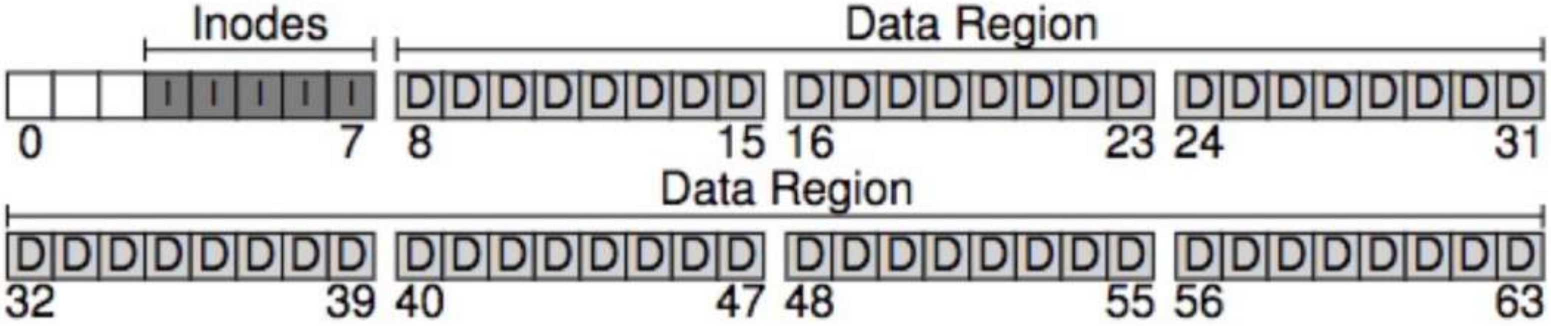
- Answers the *“how to store file information”* question

- Inode → The number corresponds to a data structure consisting of :
  - Type (e.g., file or directory)
  - Size
  - Number of data blocks
  - Address of data blocks
  - User permissions (e.g., root, etc.)
  - …

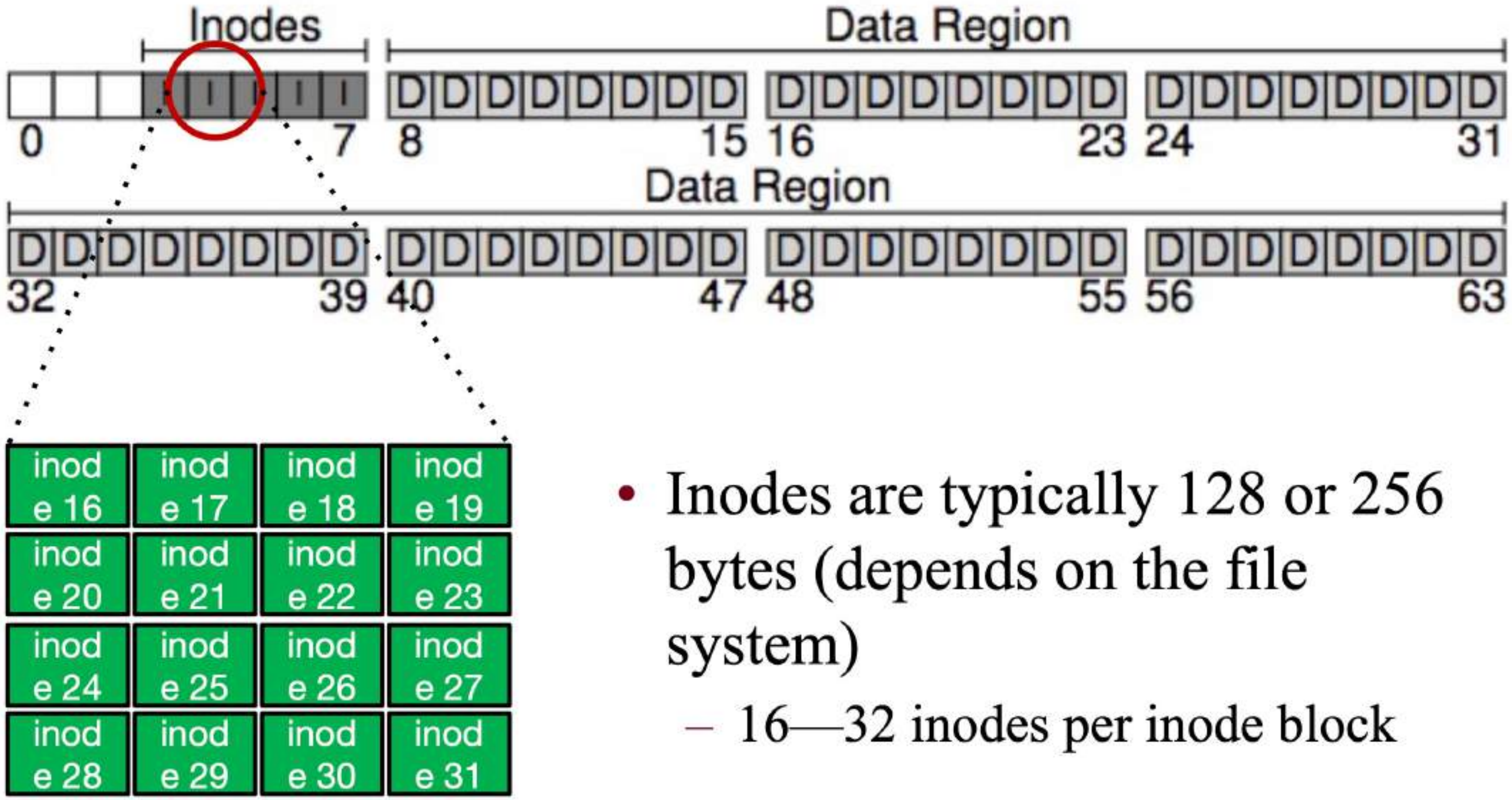
- Inodes are 128/256 bytes, depending on the FS implementation

An **inode table**

- Answers the *"where to store file information"* question
- In a part of the reserved disk space, as a table/array format

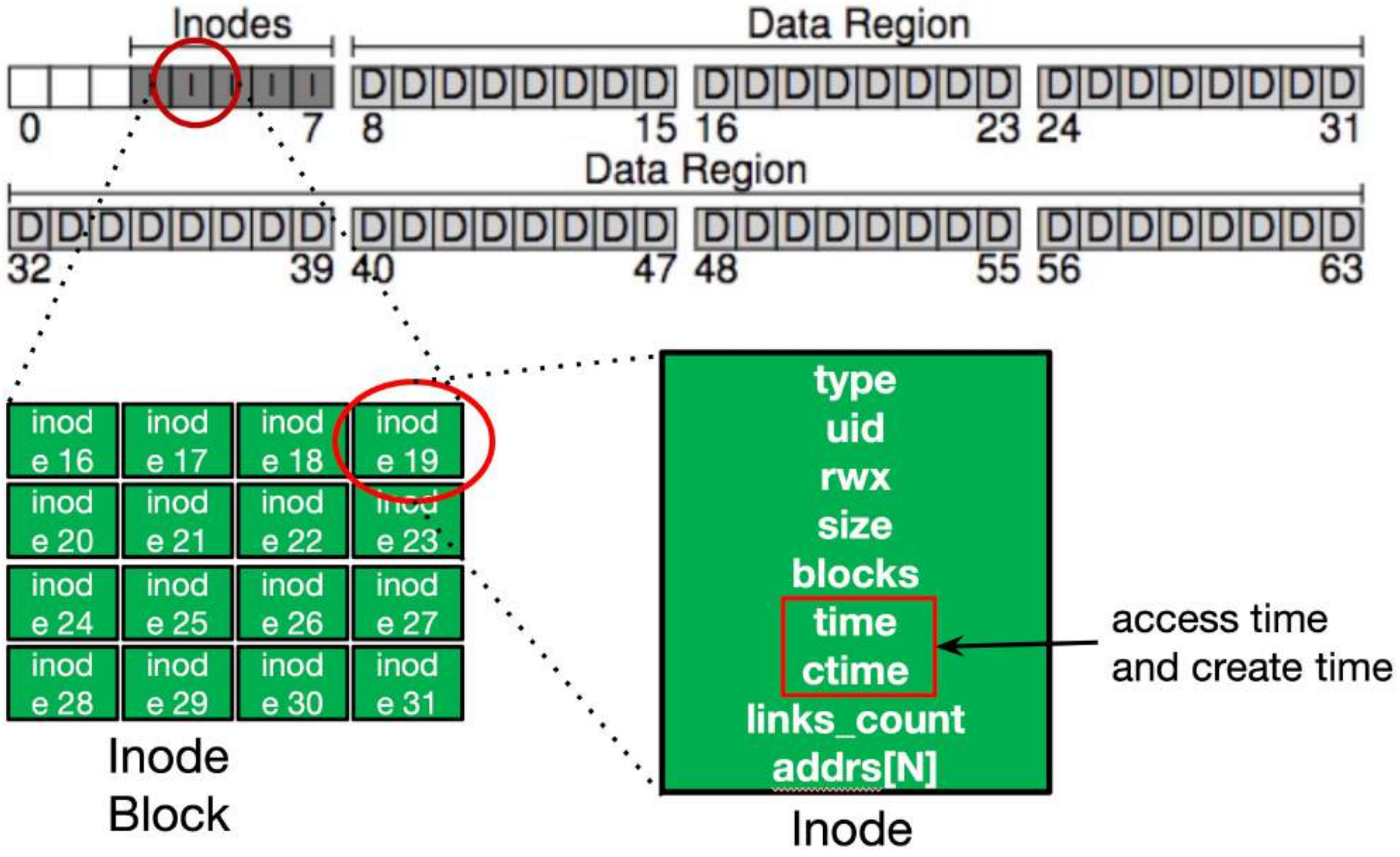


Visualizing the **inode** and its **table** in the disk

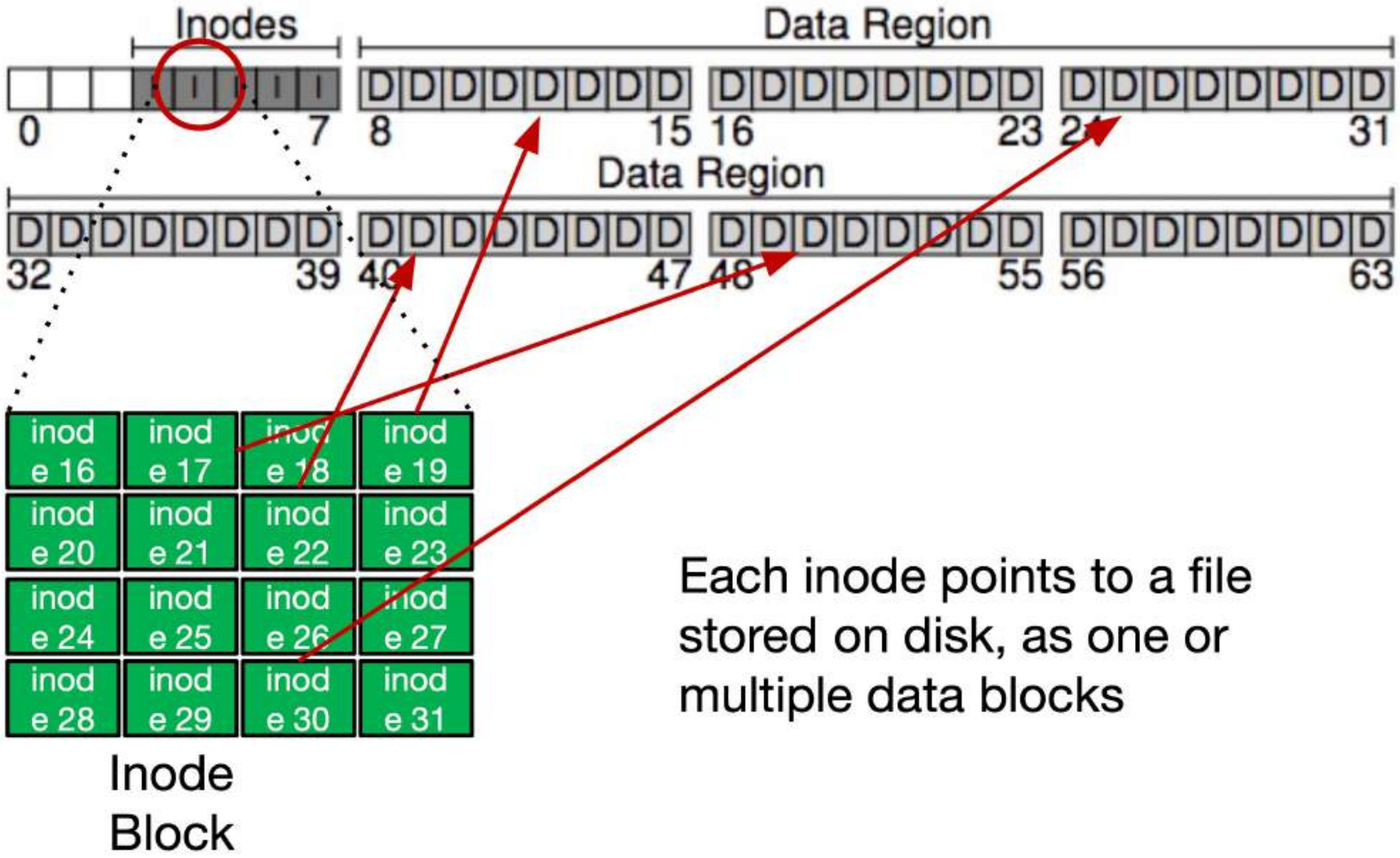


- Inodes are typically 128 or 256 bytes (depends on the file system)
  - 16—32 inodes per inode block

Visualizing the **inode** and its **table** in the disk



Visualizing the **inode** and its **table** in the disk



Each inode points to a file stored on disk, as one or multiple data blocks

Now, we must answer:

- Where to store the files in the disk?
- Where/how to store the information about existing files (e.g., inodes, path, blocks, etc.) in the disk?
- How to quickly find free locations in the disk for new files?

This question has two sub-questions. What are they?

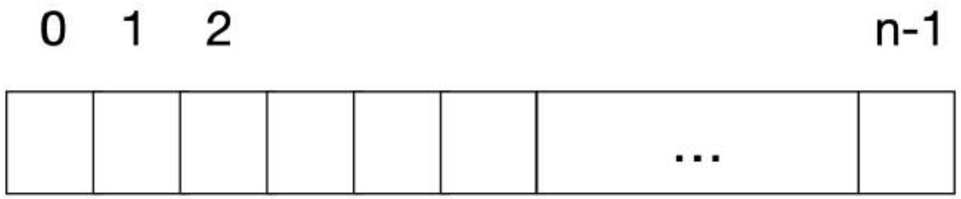
- How to quickly find free locations in the disk for new files?

1) How to quickly find free “data blocks”?
2) How to quickly find free “inodes”?

Using **bitmaps** to store free data and inode blocks

- **Can anyone tell me what's a bitmap?**

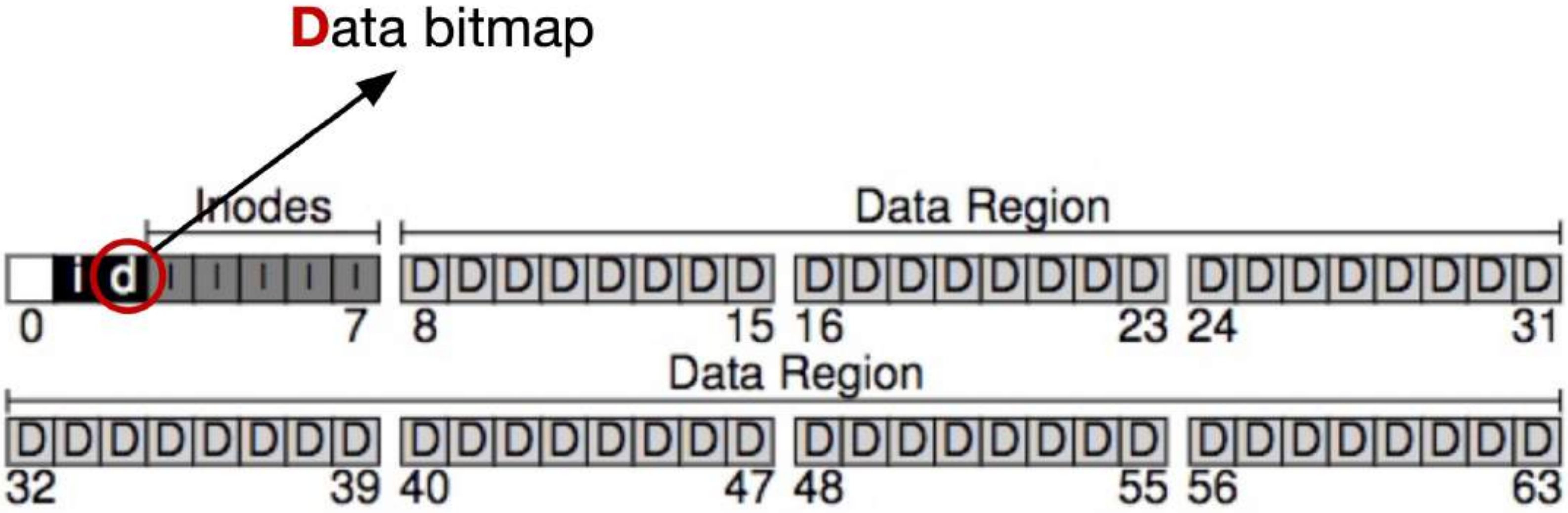
Each bit of the bitmap is used to indicate whether the corresponding object/block is **free** (0) or **in-use** (1)



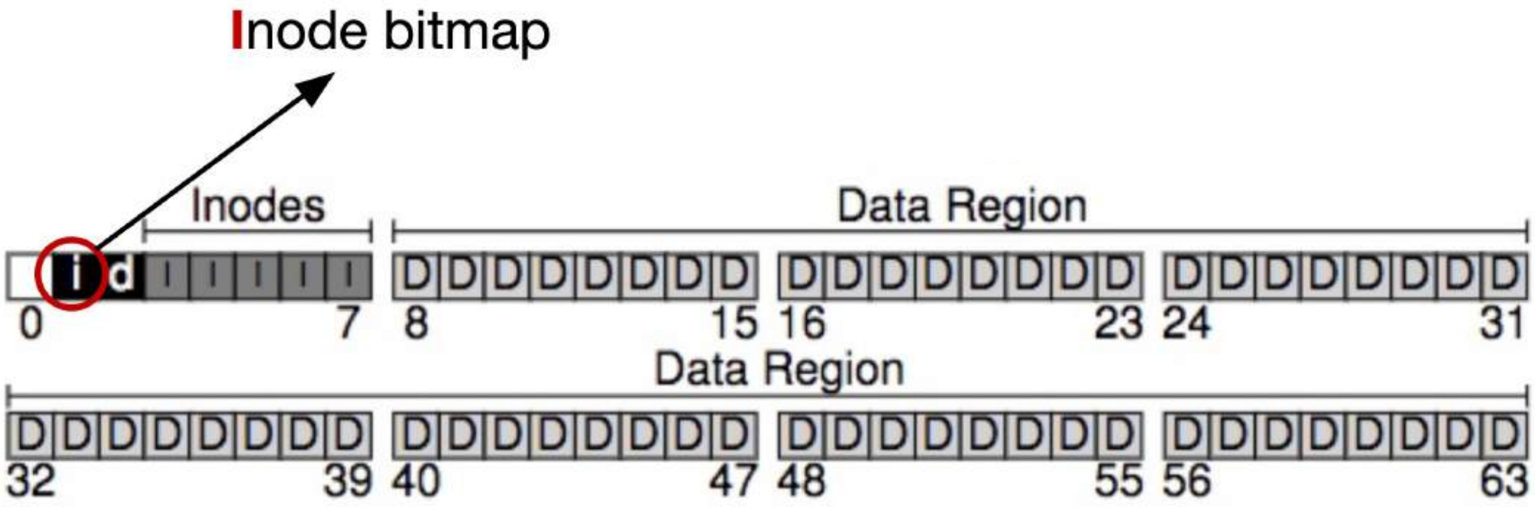
bit[i] = 1 ⇒ object[i] in use

0 ⇒ object[i] free

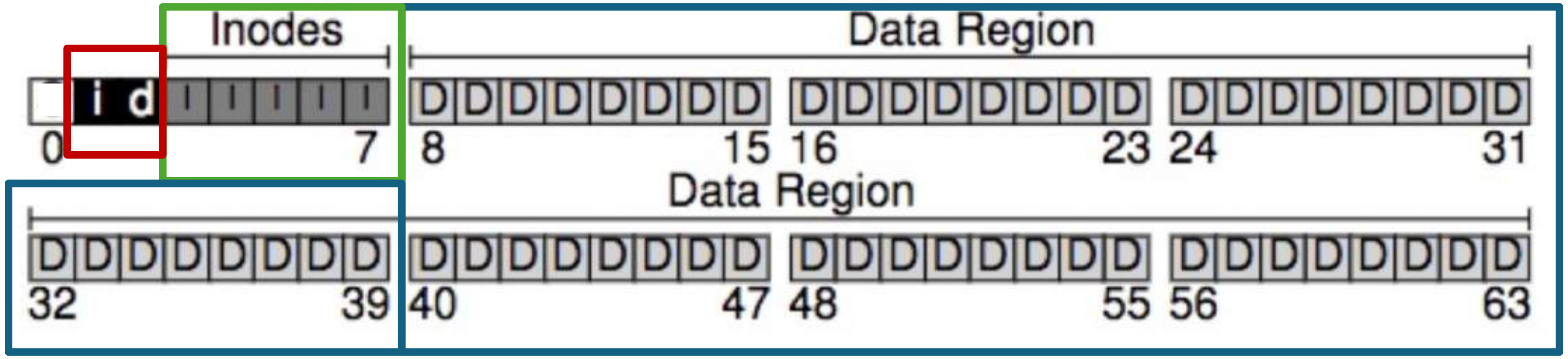
Bitmap to find free data blocks: data bitmap


Data bitmap
Inodes
Data Region
i d I I I I I
D D D D D D D D D D D D D D D D D D D D D D D D
0
7
8
15
16
23
24
31
Data Region
D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
32
39
40
47
48
55
56
63

Bitmap to find free data blocks: inode bitmap



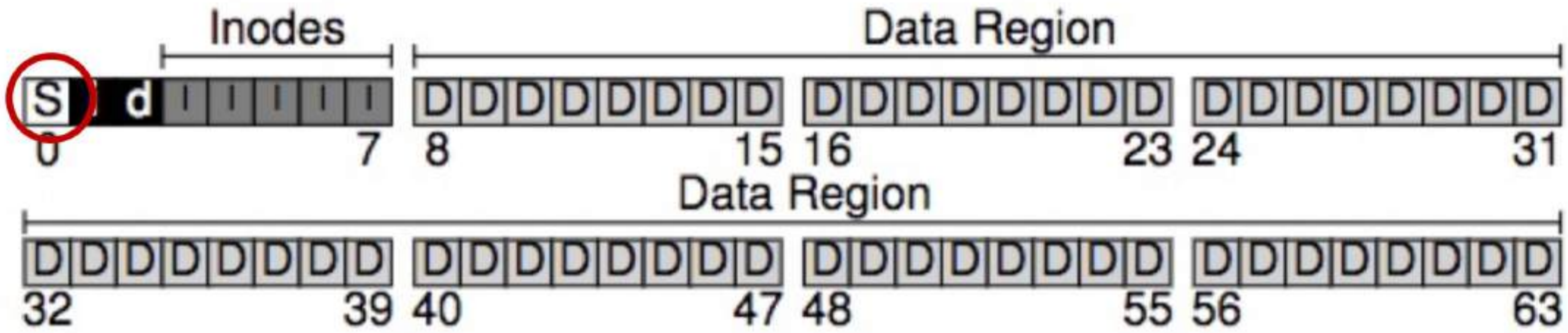
Is there another question that we may have missed?



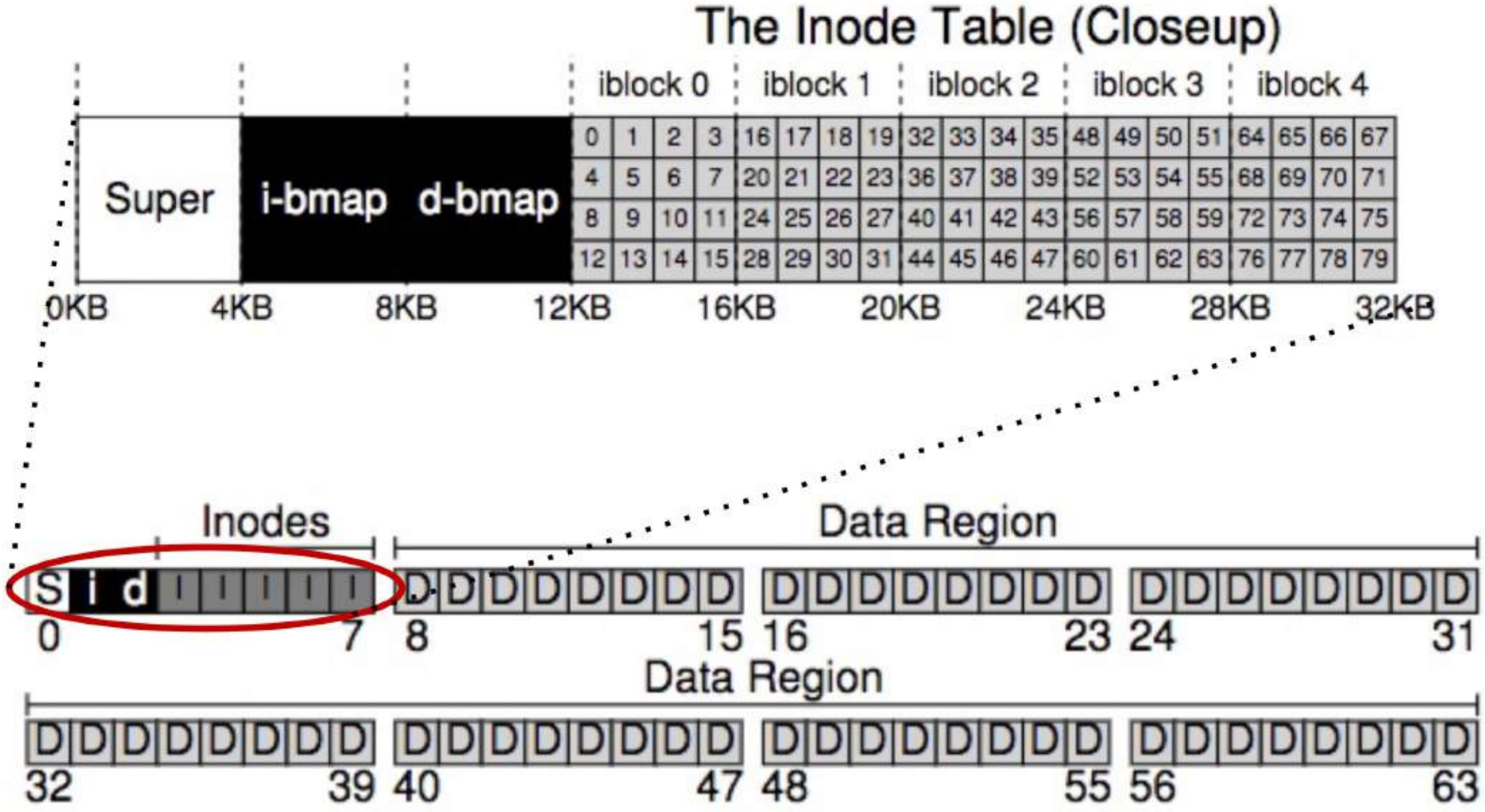
- How to persistently keep track of which regions contain all these disk structures (e.g., data blocks, inode table, bitmaps, etc.)?
    - Their size may be variable (e.g., depending on disk)

The **“superblock”**

- Keeps track of basic filesystem information, like
  - Block size
  - How many inodes are there?
  - How much space is free?



Putting it all together



Questions? Otherwise, see you next class!