Operating Systems

Eric Lo

10 - File Management

File System

- A way that lays out how data is organized on a storage device

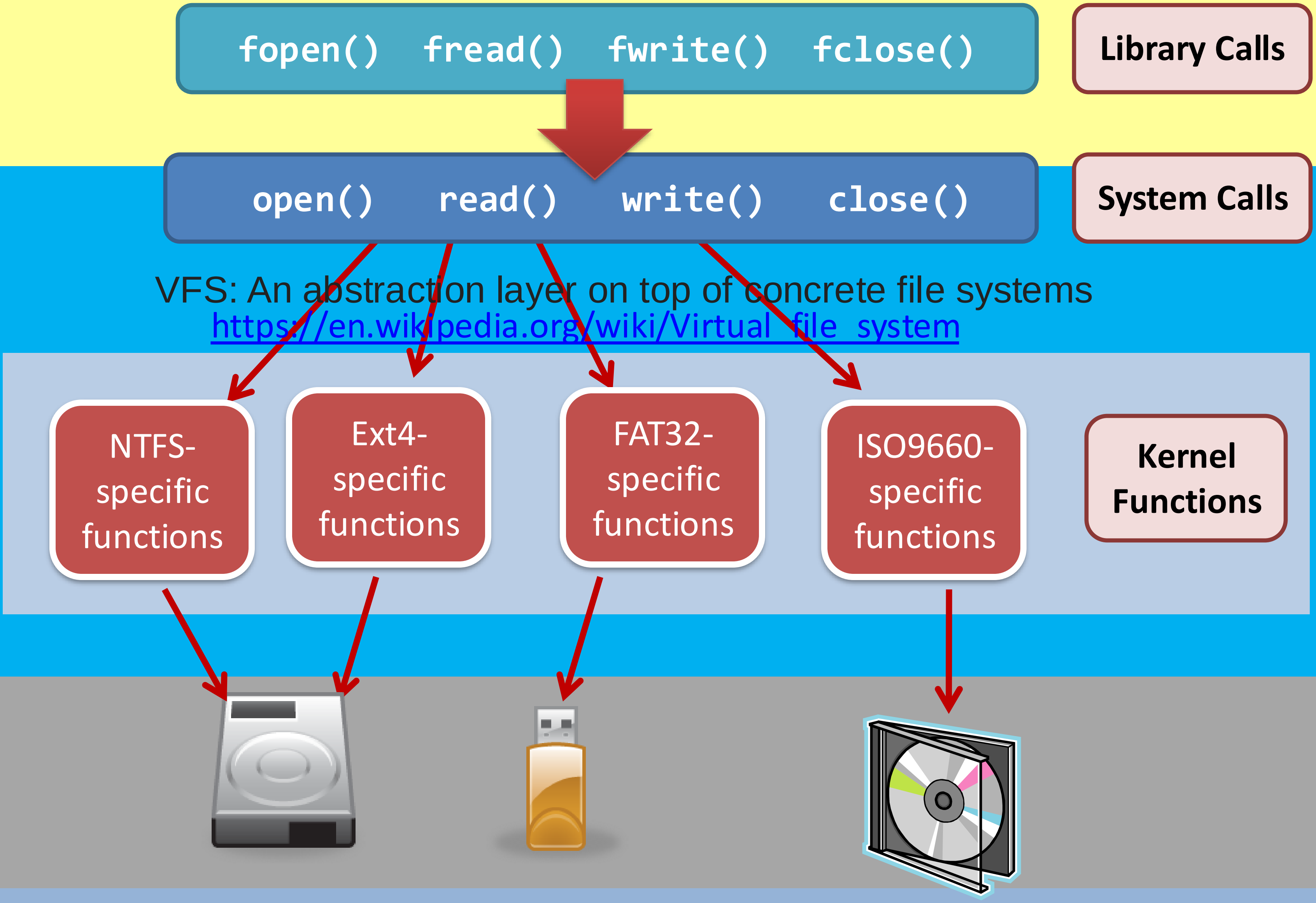
vs my desktop/your desktop

File System

- A way that lays out how data is organized on a storage device

/Assignments	vs	/3150
/3150		/A1
/3170		/A2
/3180		
/Projects		/3170
/3150		/A1
/3170		/Project
/3180		

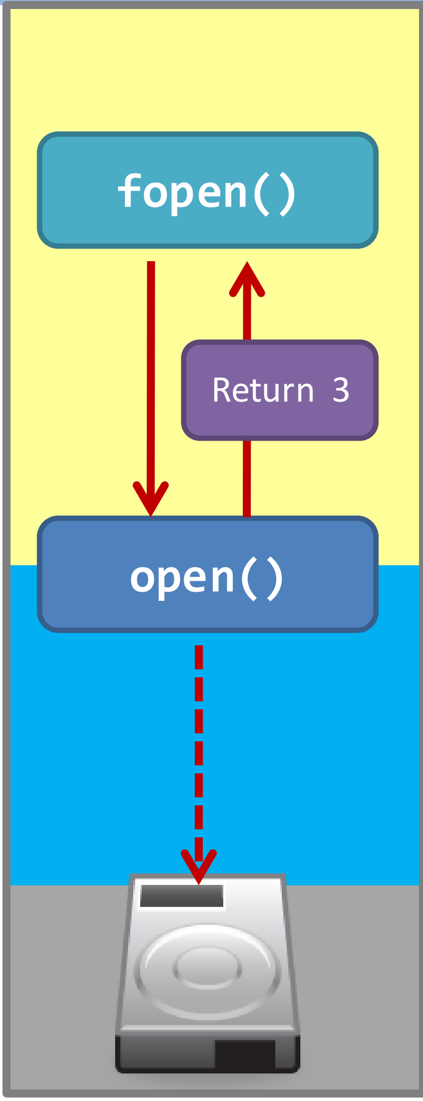
Virtual File Systems (VFS)



VFS: An abstraction layer on top of concrete file systems
https://en.wikipedia.org/wiki/Virtual_file_system

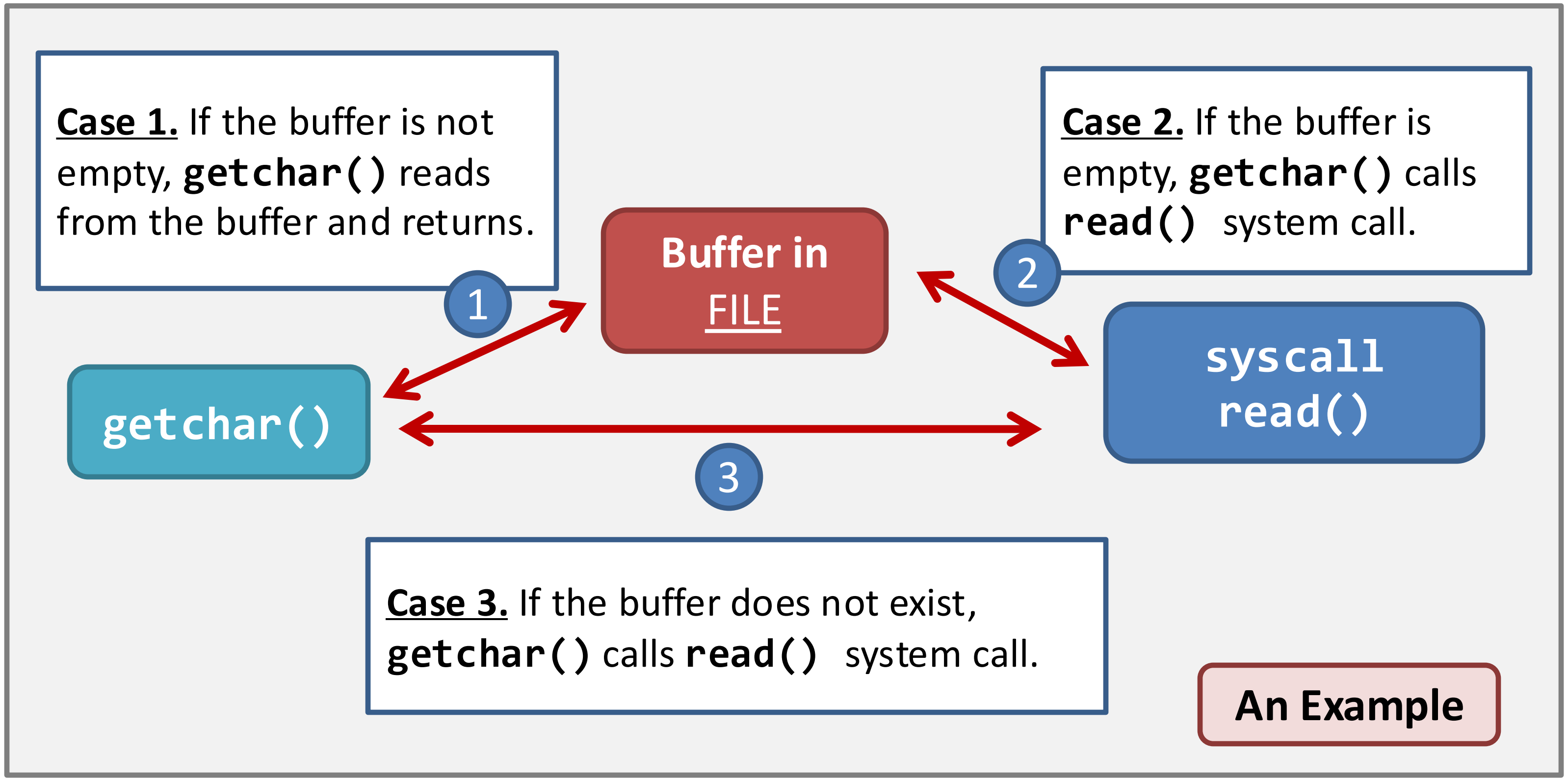
Library call VS System call

- What is **fopen()**?
  - Invokes POSIX **open()** system call
  - Returns you a pointer to **FILE**
    - "**FILE**" is a structure defined in "**stdio.h**".
  - **FILE**
    - has a userspace memory buffer
      - Provided so-called "Buffered I/O"
    - maintains where you are up to
    - internally represent your hello.txt as a number
      - The number is called a file descriptor
      - E.g., "3"



Buffered I/O and the "`FILE`" structure?

- **<u>Memory buffer</u>** in the <u>`FILE`</u> structure
  - **Reduces the number of system calls**



<u>Case 1.</u> If the buffer is not empty, **`getchar()`** reads from the buffer and returns.

<u>Case 2.</u> If the buffer is empty, **`getchar()`** calls **`read()`** system call.

<u>Case 3.</u> If the buffer does not exist, **`getchar()`** calls **`read()`** system call.

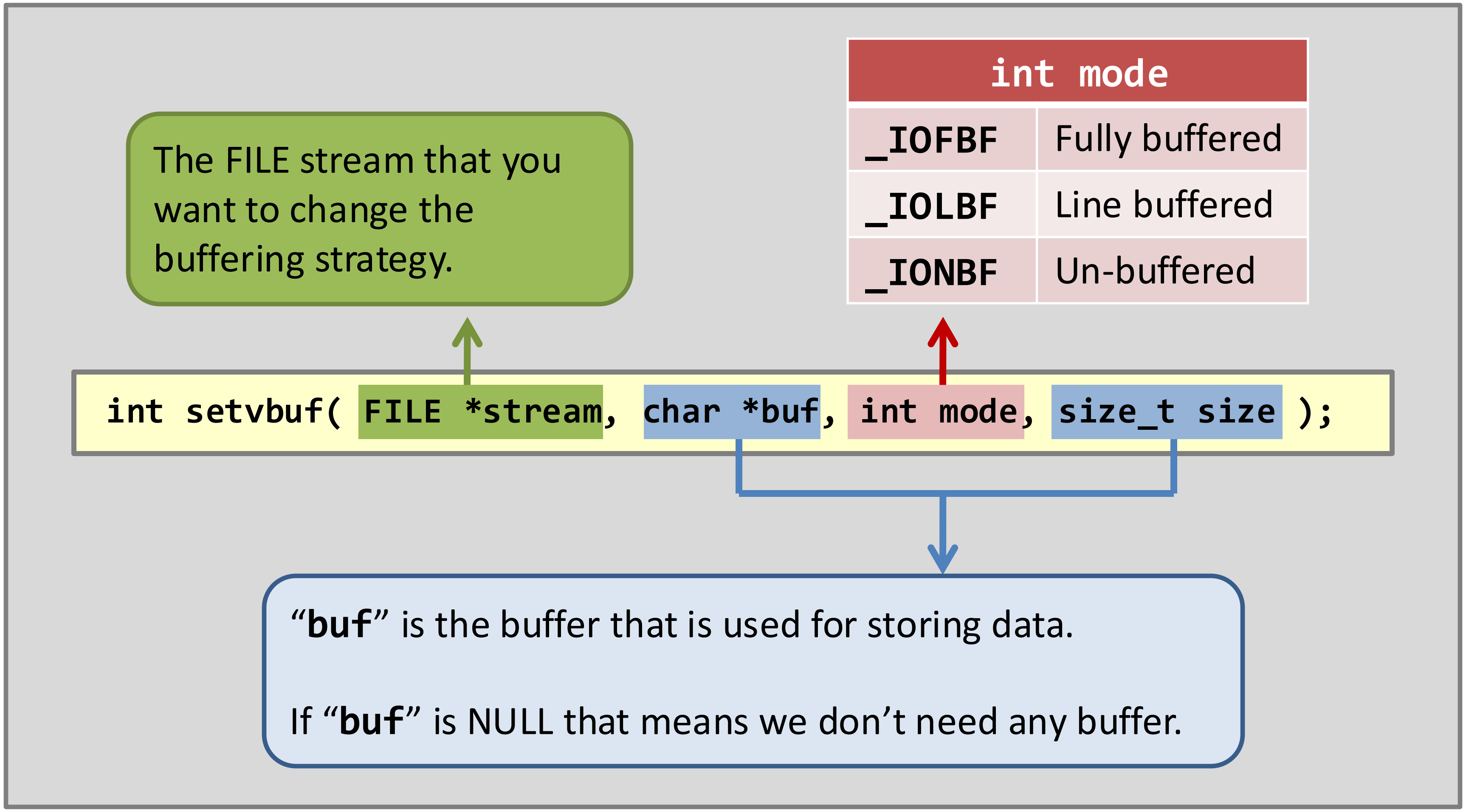
Buffered I/O – different modes

- 3 modes:

Modes	Read-related call e.g., getchar()	Write-related call e.g., putchar()
Fully-buffered	Data is **read in one bulk** and is stored in the buffer. Invoke the **`read()`** system call when the buffer becomes empty.	Data is written to the buffer. Invoke the **`write()`** system call when the buffer becomes full, or before the process terminates.
Line-buffered	Data is read into the buffer until the **newline character** is encountered.	Data is written to the buffer. When a **newline character** is encounter, **`write()`** system call is invoked.
Un-buffered	Directly translate every library call into a **`read()`** system call.	Directly translate every library call into a **`write()`** system call.

Buffered I/O – change the buffer

- **`setvbuf`**



```
int setvbuf( FILE *stream, char *buf, int mode, size_t size );
```

The FILE stream that you want to change the buffering strategy.

`int mode`	
`_IOFBF`	Fully buffered
`_IOLBF`	Line buffered
`_IONBF`	Un-buffered

“**buf**” is the buffer that is used for storing data.

If “**buf**” is NULL that means we don’t need any buffer.

Buffered I/O – change the buffer

- "**`stdin`**" and "**`stdout`**" are **line-buffered** by default.
- "**`stderr`**" is **un-buffered** by default.

```
[examples@3150] cat no_buf.c line_buf.c full_buf.c
```

Library call VS System call – what is EOF?

- Do you know what EOF really is?

```
int main(void) {
    char c;
    unsigned long long count = 0;
    while(1) {
        c = getchar();
        if(c == EOF)
            break;
        else
            count++;
    }
    printf("EOF! Read %lld bytes.\n", count);
}
```

```
[examples@3150] cat getchar_eof.c | ./getchar_eof
```

File Reading using Sys Call Directly

- You can't find any "*EOF character*" when using **system calls directly**.

```
int main(void) {
    int ret;
    char c;
    unsigned long long count = 0;
    while(1) {
        ret = read(fileno(stdin), &c, 1);
        if(ret == 0)
            break;
        else {
            count += ret;
            if(c == EOF)
                printf("WoW!\n");
        }
    }
    printf("Read %lld bytes.\n", count);
}
```

Returns 0 if no more bytes left

Any "**WoW!**"?

```
[examples@3150] cat getchar_eof.c | ./read_eof
```

Library call VS System call – what is EOF?

- Somewhere inside "**`/usr/include/stdio.h`**":

```
#ifndef EOF
# define EOF (-1)
#endif
```

- That means:
  - **No EOF character in any files!**
  - **EOF is** created by C library for you
    - functions like "**`fread()`**" *memorize* whether the end of file is reached or not!
    - If yes, it just returns -1 (EOF)! [All characters are +ve]
    - If no, it either reads data from the buffer or system calls.

Linux: Everything is a file (thing=resource)

- Regular File
  - /home/CSIC3150/helloworld.c
- Directory
  - /home/CSCI3150
    - A directory is physically a file
      - which lists the files/directories it contains
- Block special file
  - /dev/disk0
    - Binary, read/write block by block
    - Can random access
- Character special file
  - /dev/mouse
    - Binary, read/write byte by byte (a stream of "characters")
    - Can you random seek a mouse?

A “File”

- Contains two kinds of data: **attributes** and **data**.
- Attributes:
  - File size
  - File permission
  - …

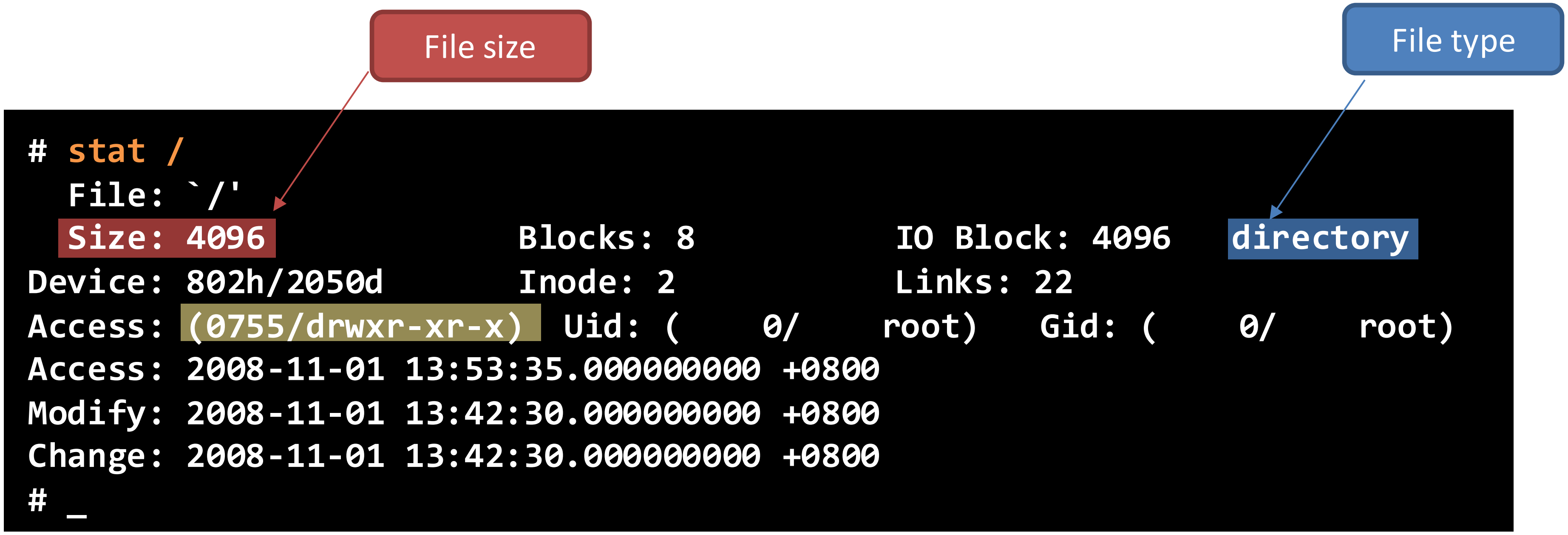
The design of FAT32 does not include any security ingredients.

Common Attributes	FAT32	NTFS	Ext2/3/4
Name	✓	✓	✓
Size	✓	✓	✓
Permission		✓	✓
Owner		✓	✓
Access, creation, modification time	✓	✓	✓

Reading attributes

- The command is **stat**. You can find:
  - type, size, permission, etc.

- The system call counterpart includes:
  - **stat()**, **fstat()**, and **lstat()**.



```
# stat /
  File: `/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 802h/2050d      Inode: 2           Links: 22
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2008-11-01 13:53:35.000000000 +0800
Modify: 2008-11-01 13:42:30.000000000 +0800
Change: 2008-11-01 13:42:30.000000000 +0800
# _
```

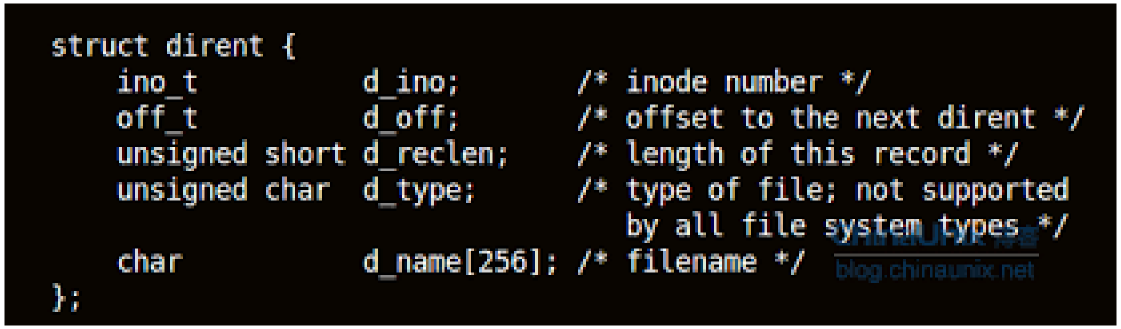
Writing attributes?

- Can you change those attributes directly?

Common Attributes	Way to change them?	
	Command?	Syscall?
Name	`mv`	`rename()`
Size	`edit it using vi and then save…`	`write(), truncate(), etc.`
Permission	`chmod`	`chmod()`
Owner	`chown`	`chown()`
Access, creation, modification time	`touch`	`utime()`

A directory

- **A directory is a file**
  - consisting of **dir**ectory **ent**ries
    - “**dirent**” is a struct



```
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 */
};
```

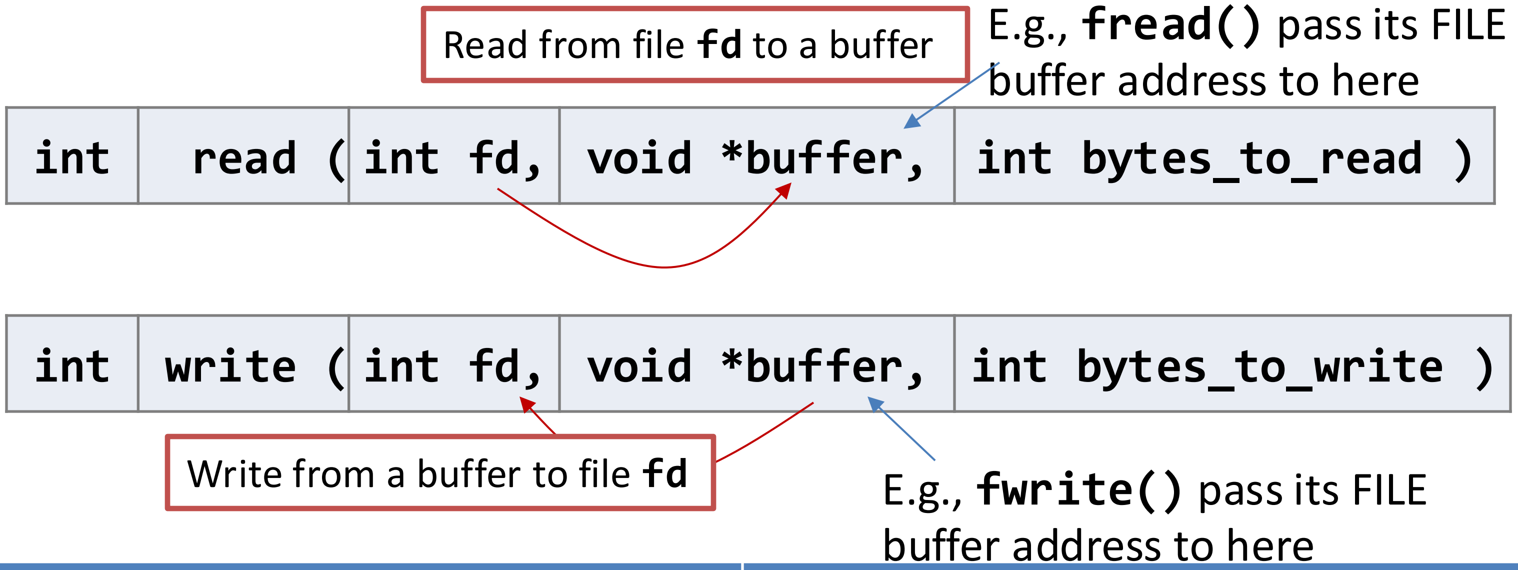
Reading a directory

```
int main(int argc, char **argv) {
    DIR *dir; //a type
    char *input = "/";
    struct dirent *entry;

    dir = opendir(input);                           // open
    while( (entry = readdir(dir)) != NULL ) {   // read
        printf("%ld\t\t%s\n",
                (long) entry->d_ino,   // unique ID
                entry->d_name);        // name, max char: 255
    }
    closedir(dir);                                  // close
}
```

```
[examples@3150] cat simple_ls.c
```

System calls: read() & write()



```
int  read ( int fd, void *buffer, int bytes_to_read )

int  write ( int fd, void *buffer, int bytes_to_write )
```

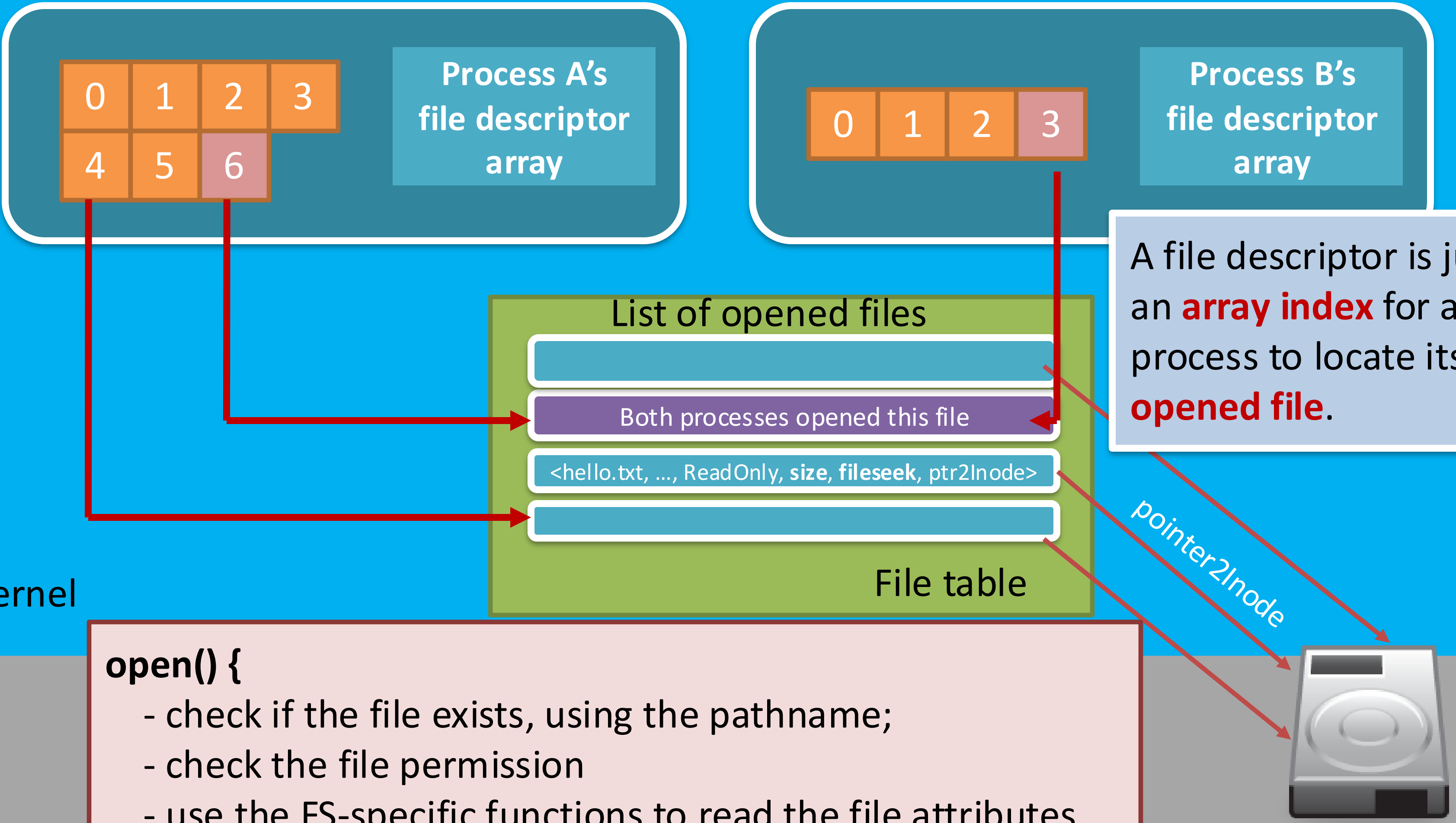
Read from file **fd** to a buffer
E.g., **fread()** pass its FILE buffer address to here
Write from a buffer to file **fd**
E.g., **fwrite()** pass its FILE buffer address to here

Library calls that eventually invoke the `read()` system call	Library calls that eventually invoke the `write()` system call
`scanf(), fscanf()`	`printf(), fprintf()`
`getchar(), fgetc()`	`putchar(), fputc()`
`gets(), fgets()`	`puts(), fputs()`
`fread()`	`fwrite()`

```
fopen()  //returning you a FILE*
```

```
FILE struct { user space buffer, fd=6}
```

```
int open() //returning a fd
```



0 1 2 3
4 5 6
Process A's file descriptor array
0 1 2 3
Process B's file descriptor array
List of opened files
Both processes opened this file
<hello.txt, …, ReadOnly, **size**, **fileseek**, ptr2Inode>
File table
pointer2Inode
Kernel

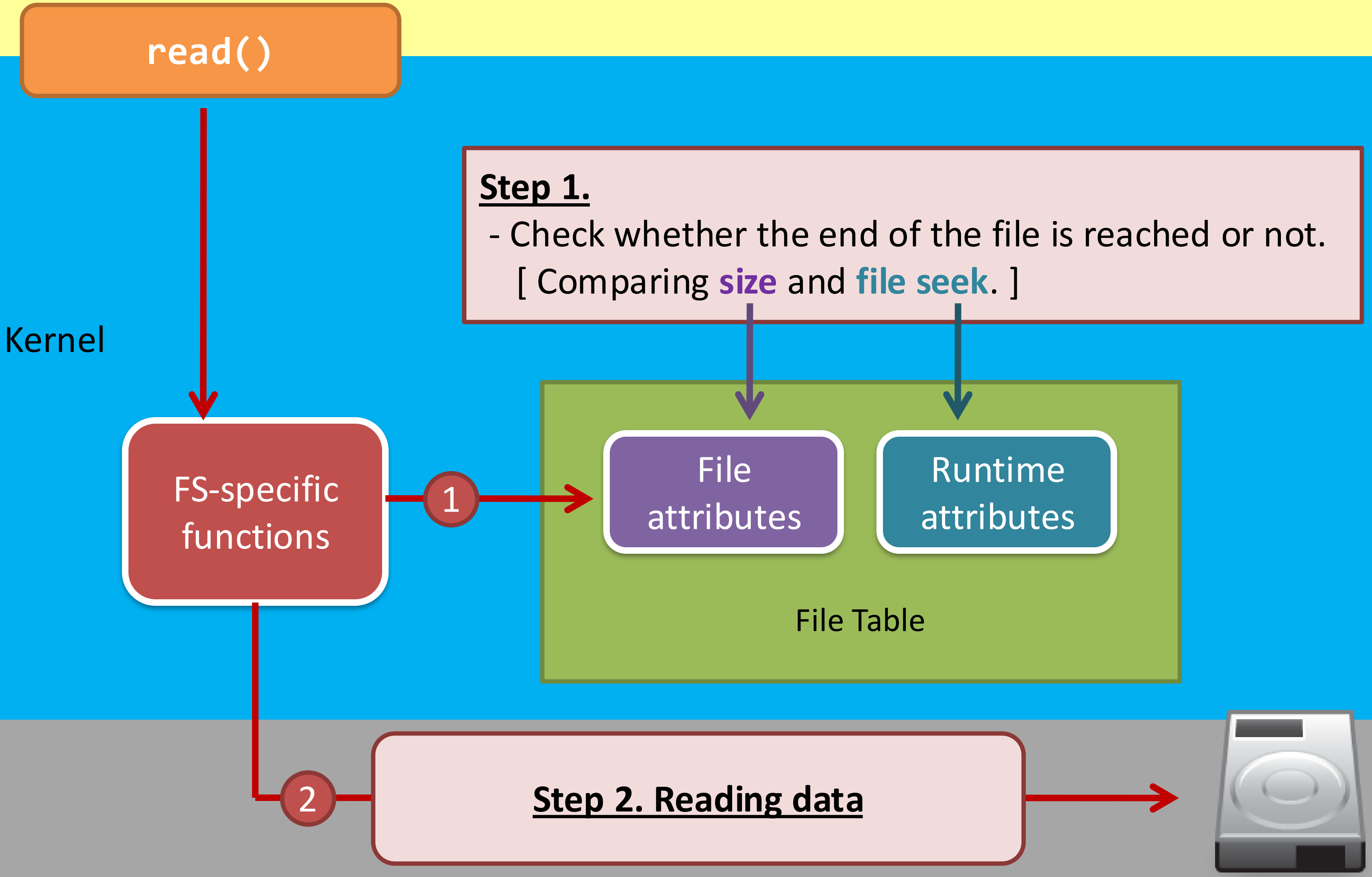
A file descriptor is just an array index for a process to locate its opened file.

open() {

- check if the file exists, using the pathname;
- check the file permission
- use the FS-specific functions to read the file attributes and store them in the kernel's file table as one entry

…

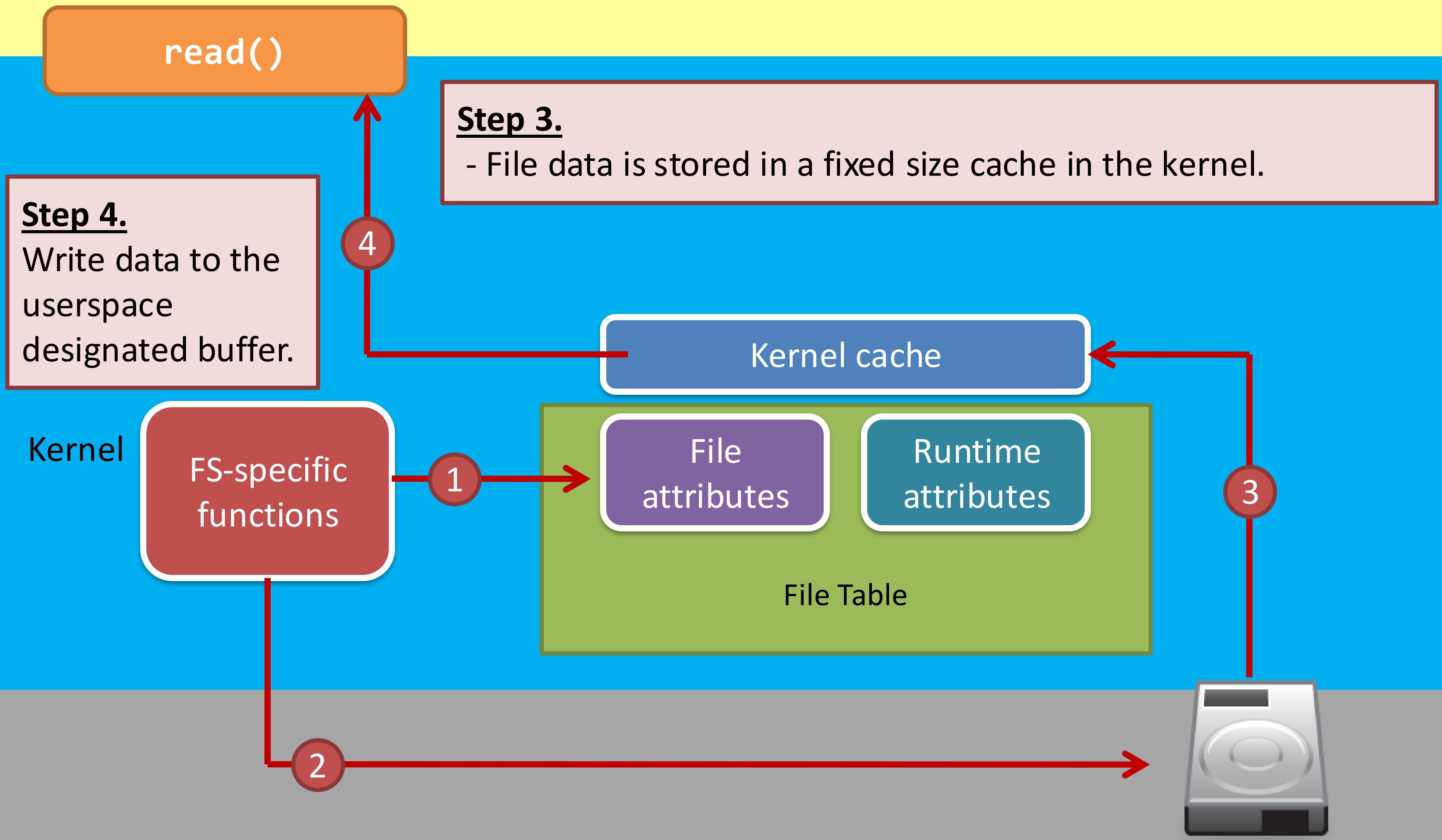
System call: read()



Step 1.
- Check whether the end of the file is reached or not.
  [Comparing **size** and **file seek**.]

Step 2. Reading data

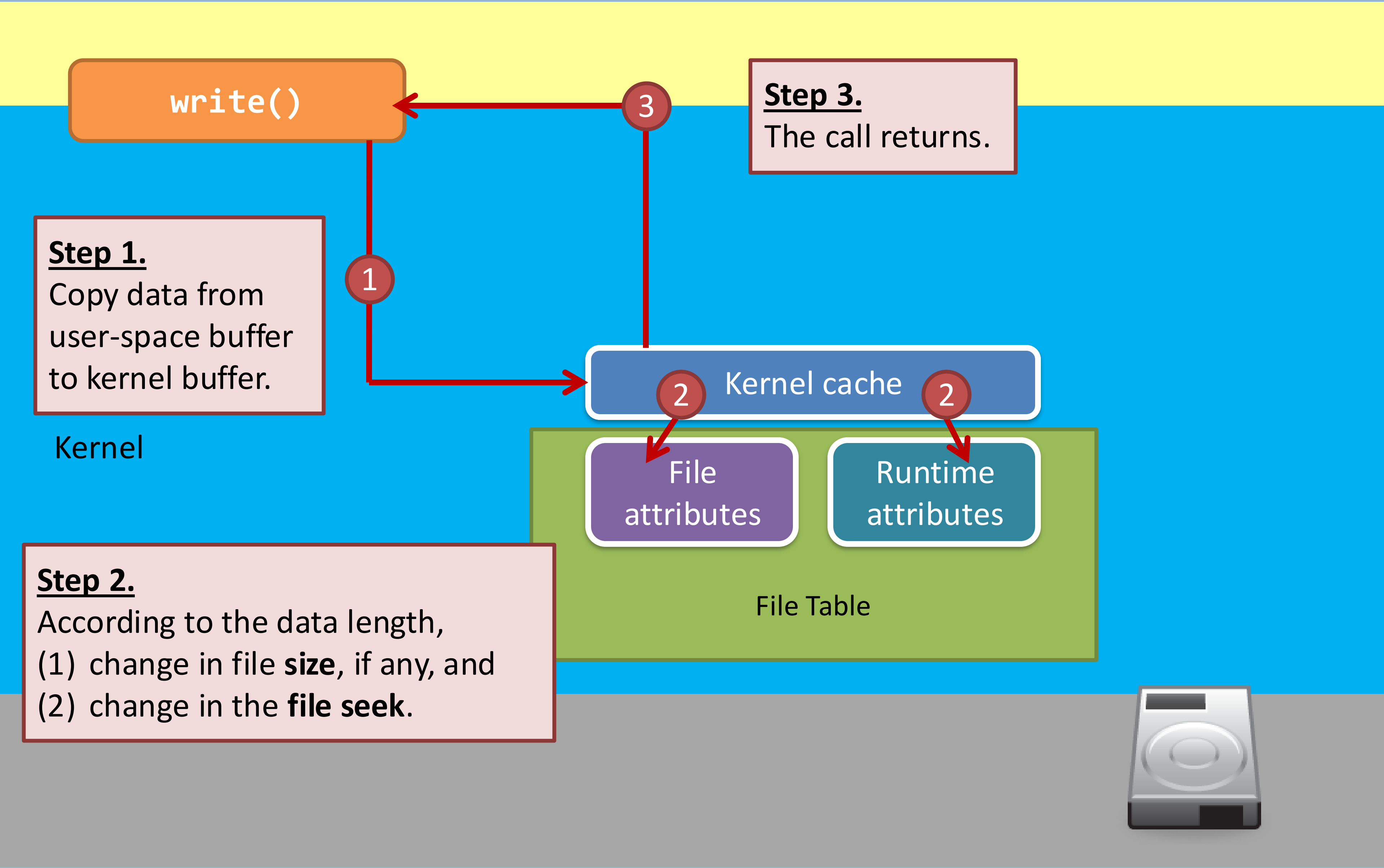
System call: read()



Step 3.
- File data is stored in a fixed size cache in the kernel.

Step 4.
Write data to the userspace designated buffer.

System call: write()

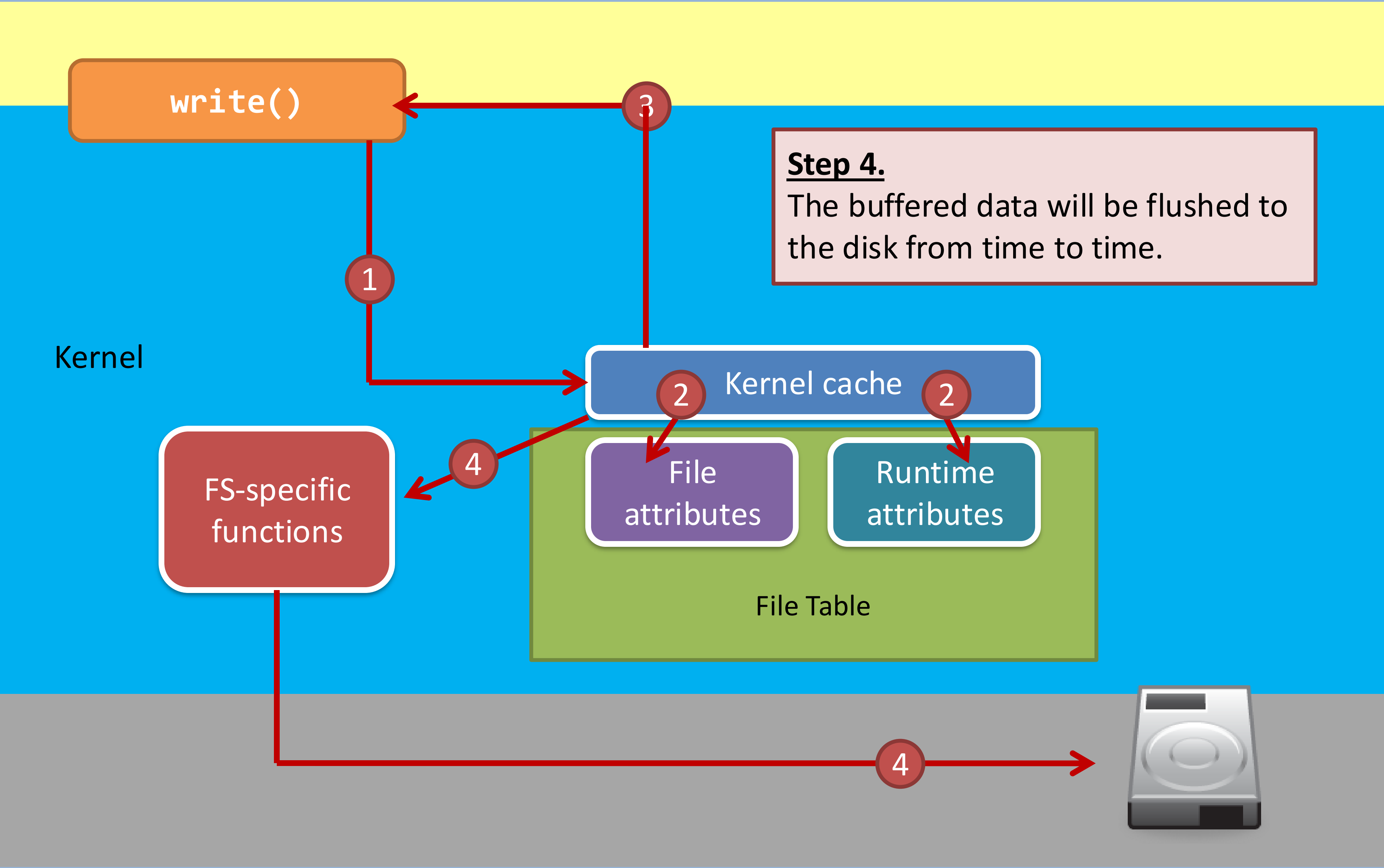


Step 1.
Copy data from user-space buffer to kernel buffer.

Step 2.
According to the data length,
(1) change in file **size**, if any, and
(2) change in the **file seek**.

Step 3.
The call returns.

System call: write()



Step 4.
The buffered data will be flushed to the disk from time to time.

The kernel buffer cache implies…

- Improving reading performance
- Improving writing performance
- So you now know why **you should not press the reset button**?
- So you now know **why you need to press the "eject" button before removing USB drives?**