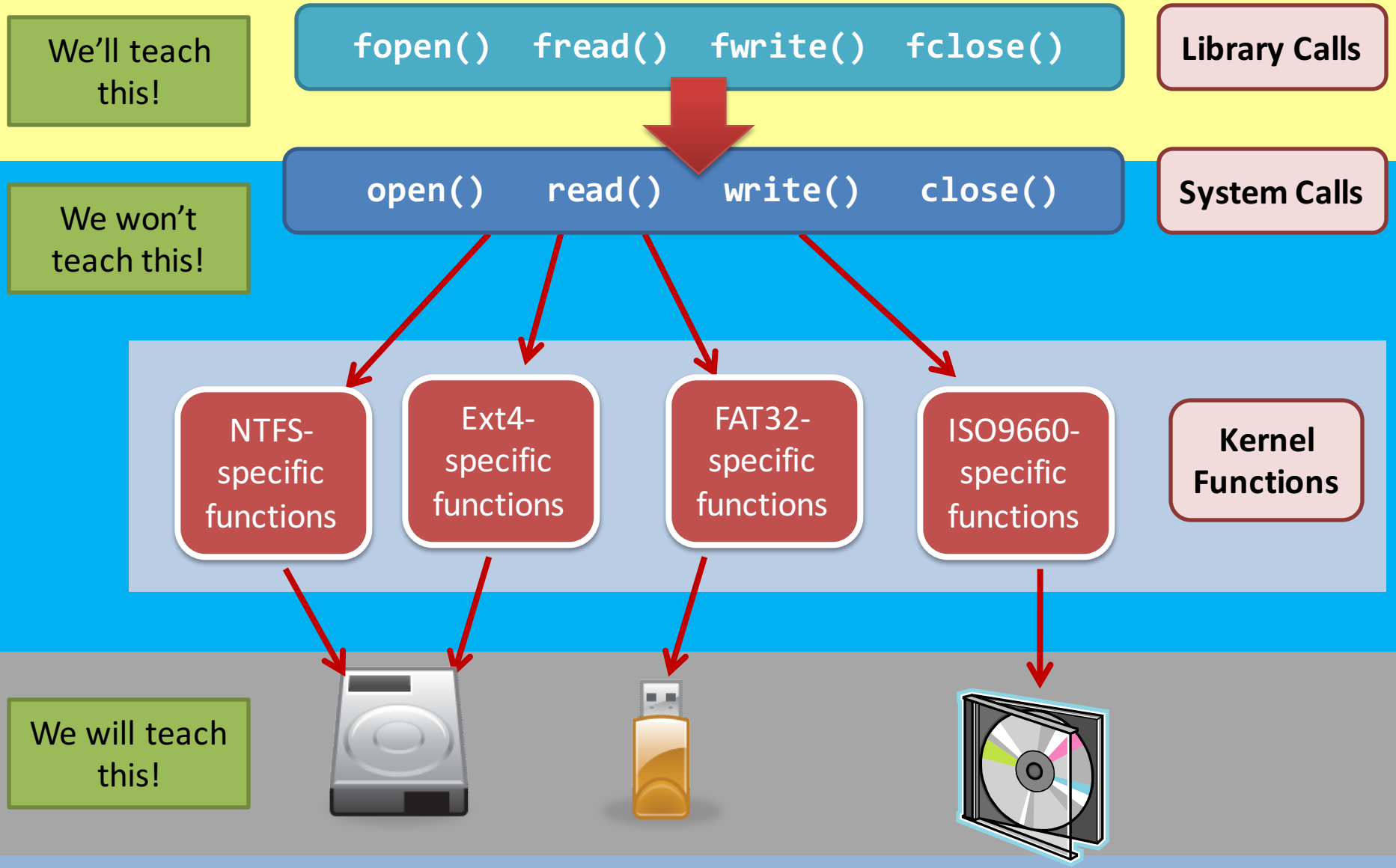
3150 - Operating Systems

Dr. WONG Tsz Yeung

Chapter 3, part 1
File Systems – Programmer Perspectives.

- Here comes the chapter that you can play with something solid, e.g., to corrupt a disk …

Outline


We'll teach this!
fopen() fread() fwrite() fclose()
Library Calls
We won't teach this!
open() read() write() close()
System Calls
NTFS-specific functions
Ext4-specific functions
FAT32-specific functions
ISO9660-specific functions
Kernel Functions
We will teach this!

Class Discussion: Question #1

OS == FS? Reasons?

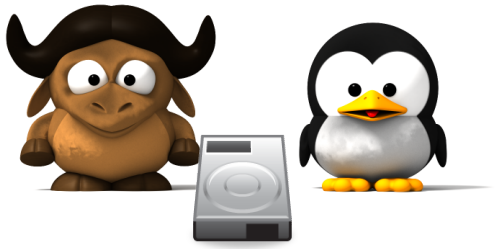
- An OS supports a FS.
- An OS supports more than one FS.
- A FS can be used by more than one OS.
- But, “FS != OS”.

Class Discussion: Question #2

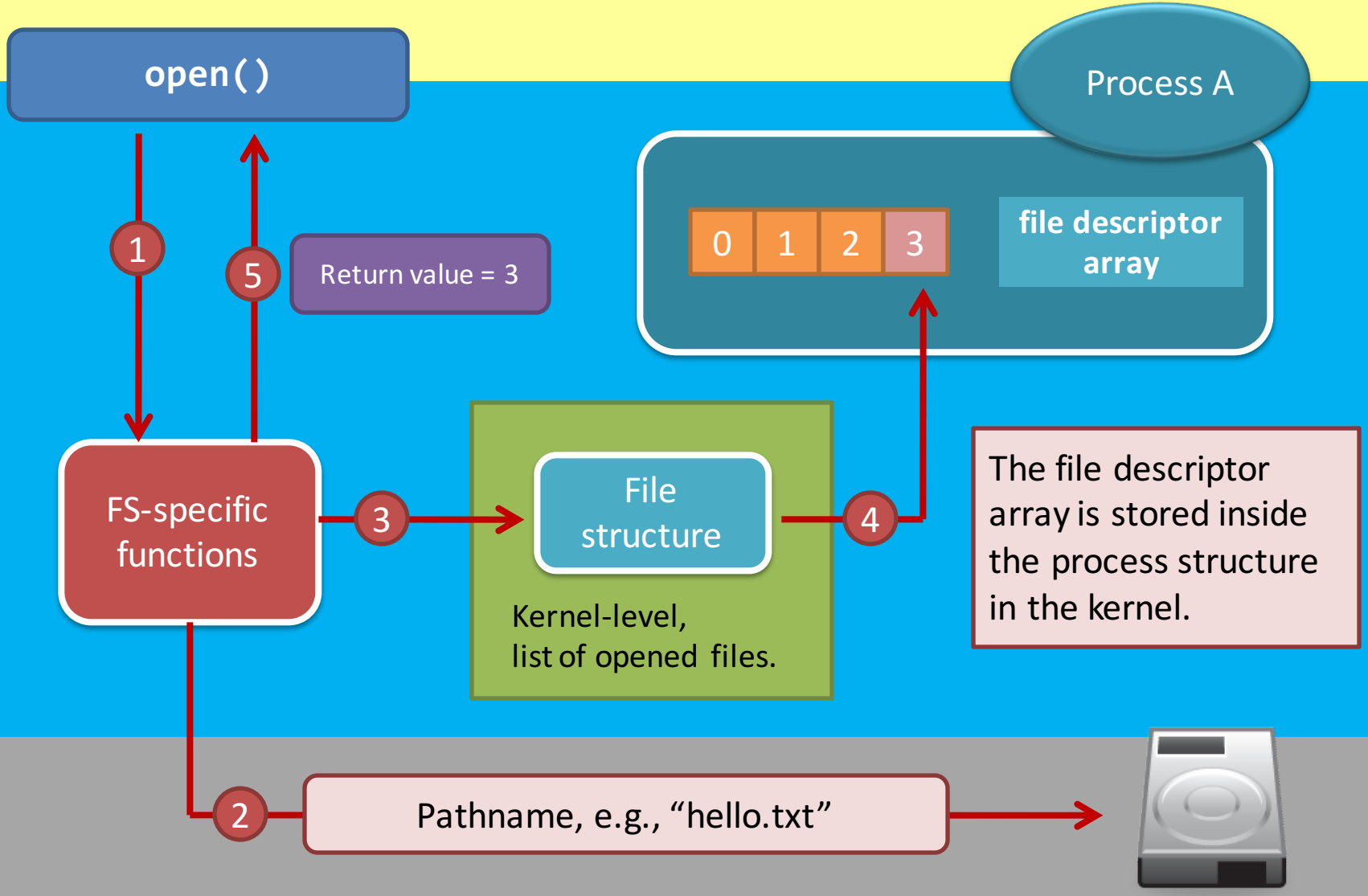
Storage Device == FS? Reasons?

- A FS needs a storage device.
  - But, a device can be either physical or virtual.
- A storage device is just a container.
  - Doesn't need to know what FSes are stored.
  - Doesn't need to know how many FSes are stored (on different partitions.
- So, "Storage Device != FS"!

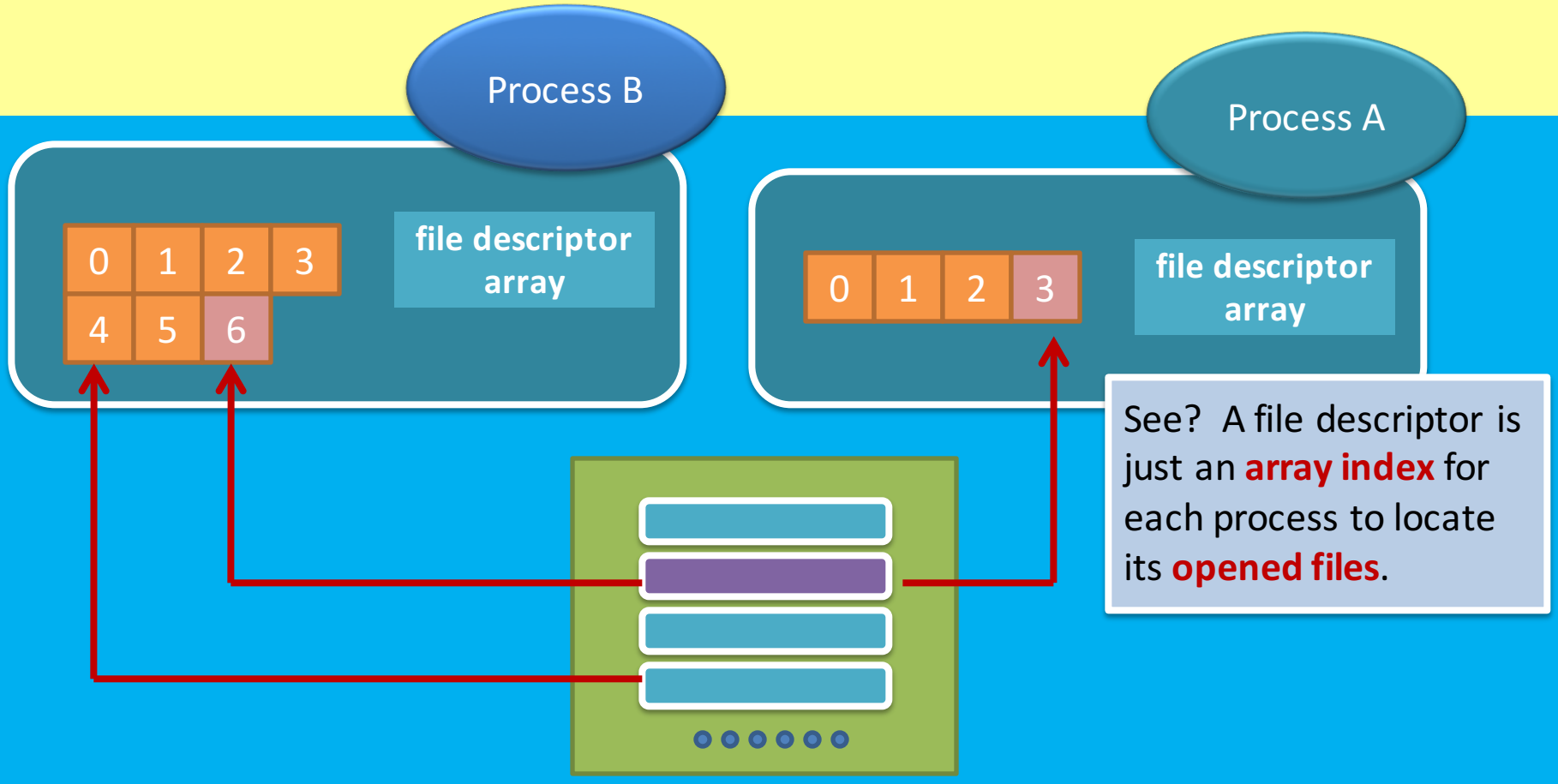
Looking at FS from the userspace
- GNU C Library call VS System call?



What is a file descriptor?



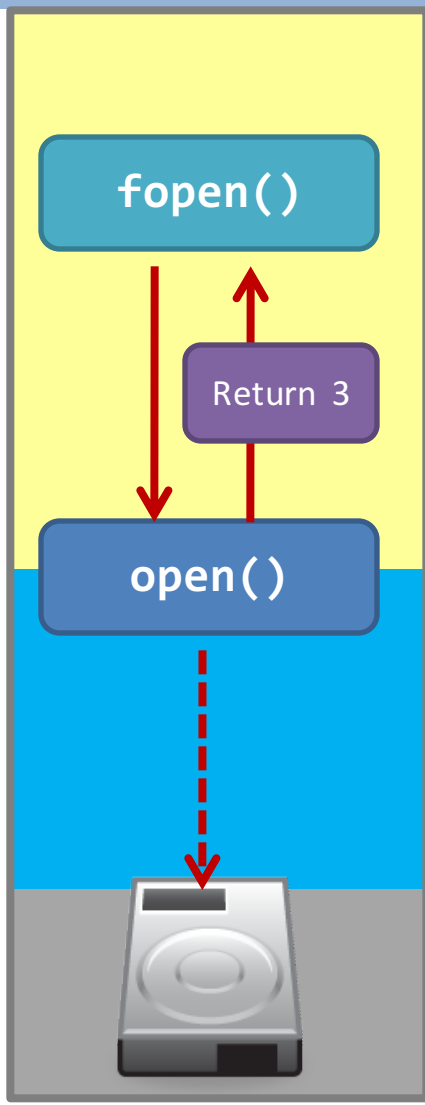
What is a file descriptor?



Although a file is opened by two different processes, the kernel uses **one structure to maintain it**!

Library call VS System call

- What is **`fopen()`**? What is the type "**`FILE`**"?
  - First thing first, **`fopen()`** calls **`open()`**.
  - "**`FILE`**" is just a structure in defined in "**`stdio.h`**".
  - However, **`fopen()`** *creates memory* for the "**`FILE`**" structure.
    - Fact: occupying space in the area of dynamically allocated memory, i.e., `malloc()`



What is inside the "FILE" structure?

- There is a lot of helpful data in **FILE**:
  - Two important things: the **file descriptor** and **a buffer**!

```
int main(void) {
    printf("fd of stdin  = %d\n", fileno(stdin)  );
    printf("fd of stdout = %d\n", fileno(stdout) );
    printf("fd of stderr = %d\n", fileno(stderr) );
}
```

fileno() returns the file descriptor of the FILE structure.

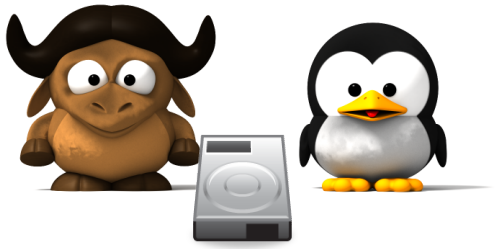
The type of **stdin**, **stdout**, and **stderr** is "**FILE ***"

```
$ ./fileno
fd of stdin  = 0
fd of stdout = 1
fd of stderr = 2
$ _
```

[examples@3150] cat fileno.c

Looking at FS from the userspace

- GNU C Library call VS System call?
  - Buffered I/O and efficiency.



read() & write()

- You know, I/O-related calls will invoke system calls.

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

From file to buffer.

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

From buffer to file.

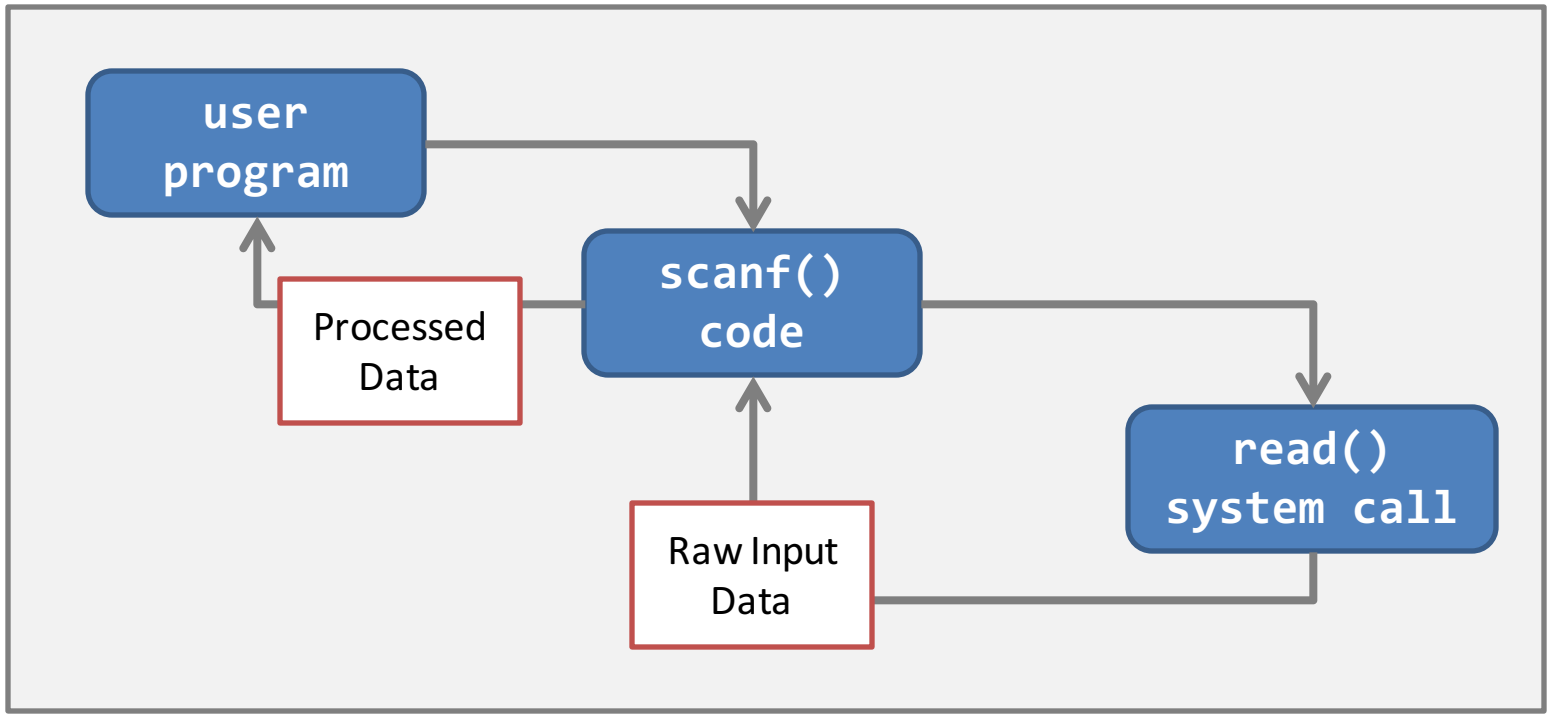
Note: I modified the function prototypes.

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()`

read() & write()

- **scanf()** as an example!

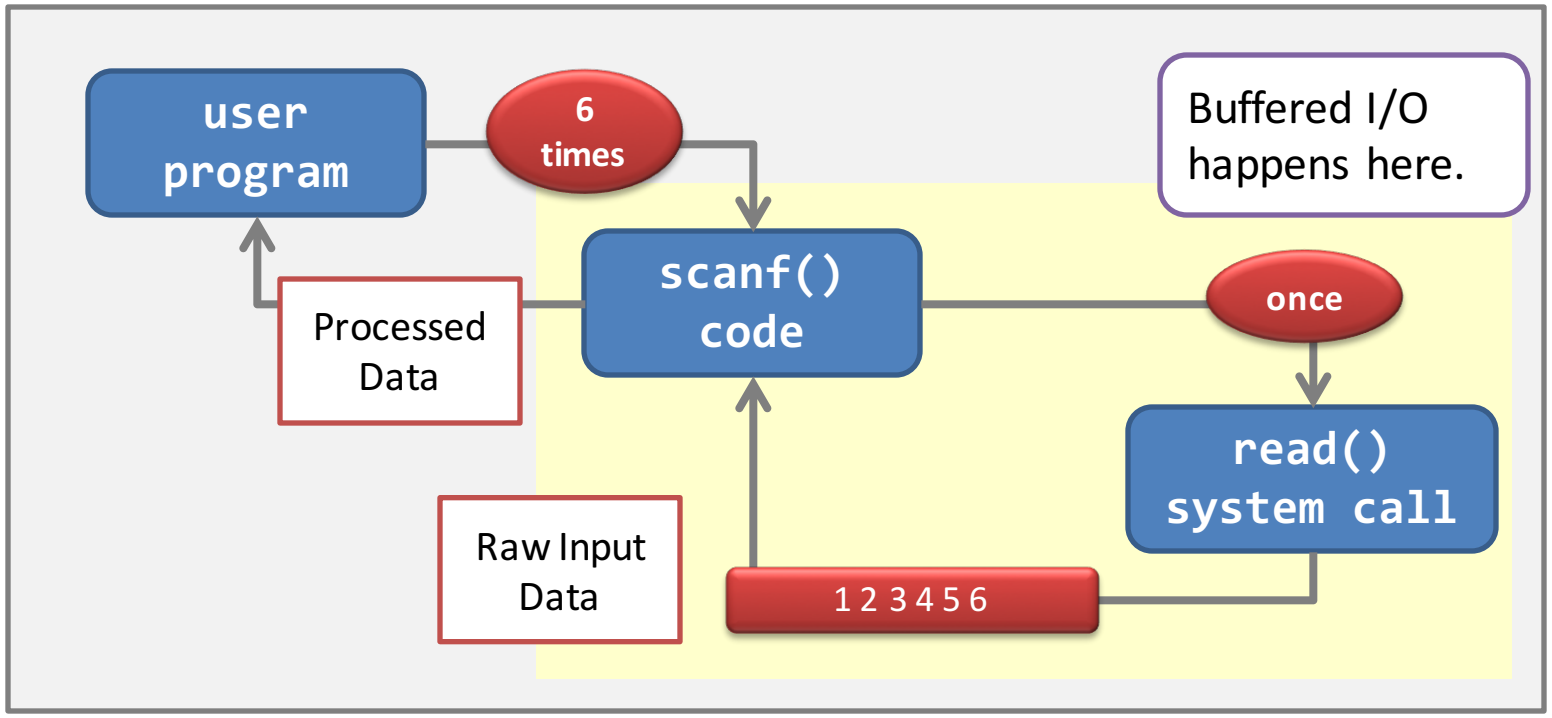
```
int main(void) {
    int input;
    while(1) {
        scanf("%d", &input);
        printf("%d\n", input);
    }
}
```



What is buffered I/O?

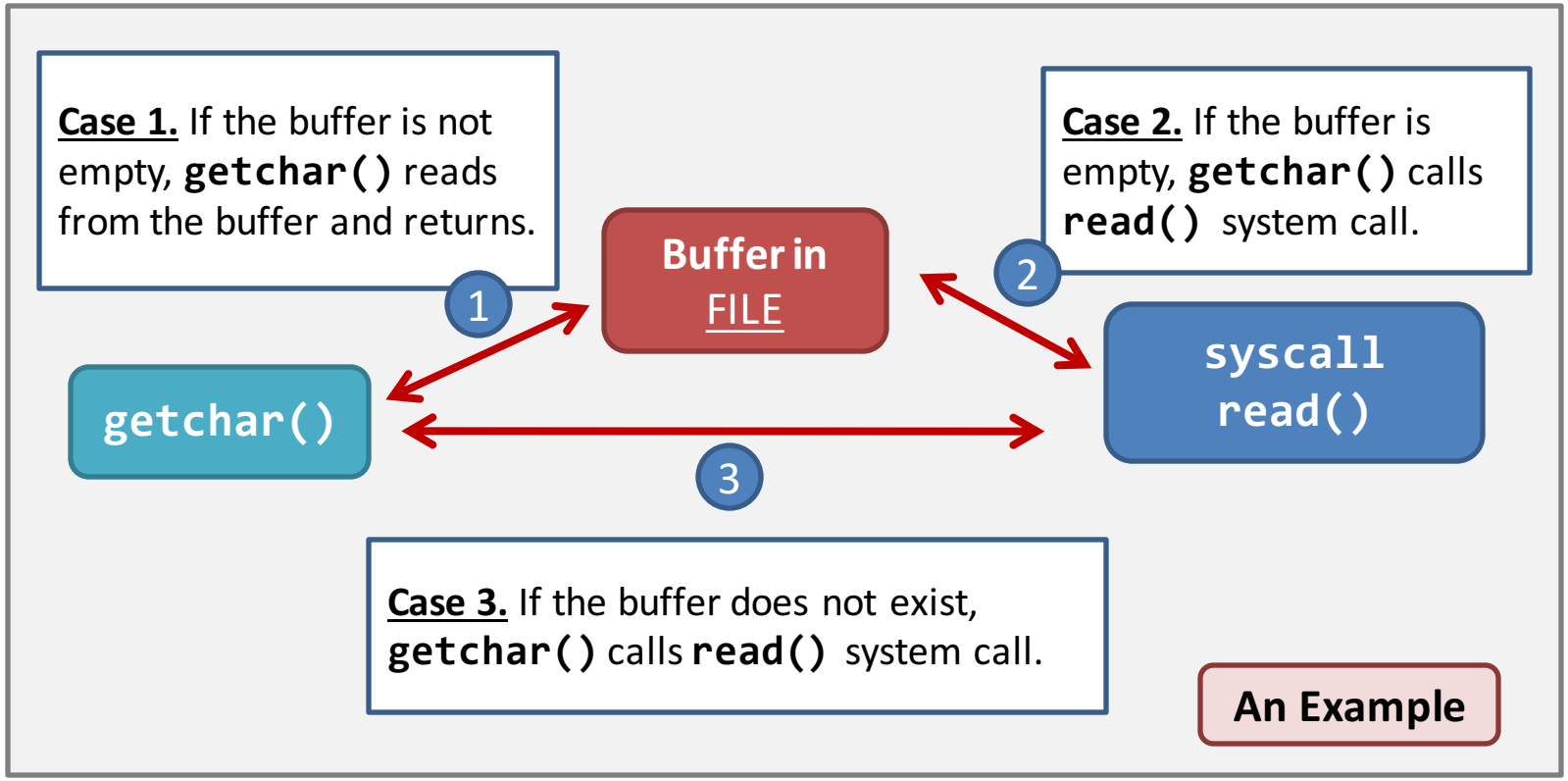
- If I input "**1 2 3 4 5 6**", you will find 6 outputs, right?
  - But, are there 6 `read()` calls?

```
int main(void) {
    int input;
    while(1) {
        scanf("%d", &input);
        printf("%d\n", input);
    }
}
```



Buffered I/O and the "FILE" structure?

- There is a **memory buffer** in the **FILE** structure and **this cache (or buffer) to reduce the number of system calls**!



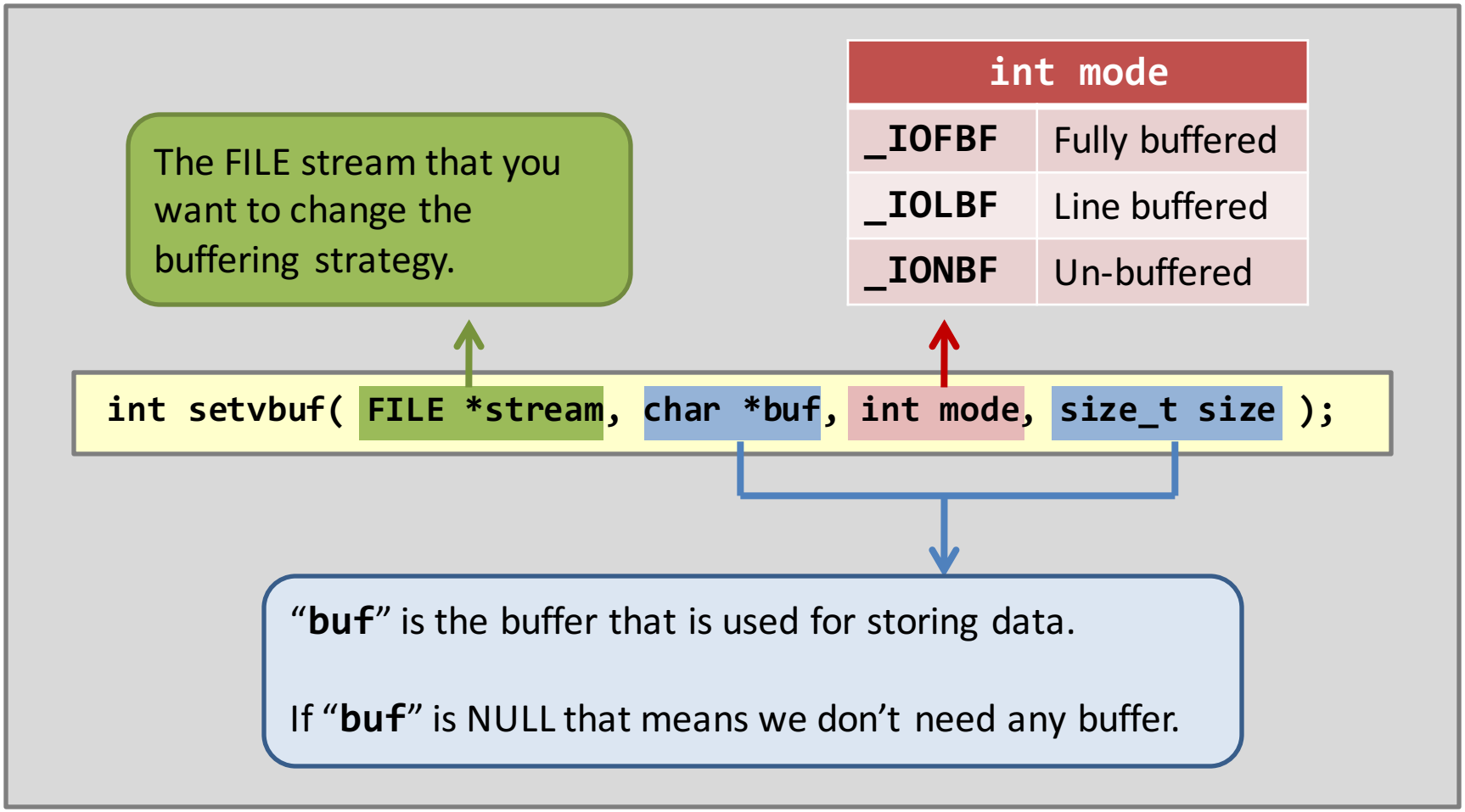
Buffered I/O – different modes

- 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

- There is a convenient call that controls **everything**!



Buffered I/O – change the buffer

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

```
char buf[1024];
int main(void) {
    ......
    printf("H");    sleep(1);
    printf("E");    sleep(1);
    printf("L");    sleep(1);
    printf("L");    sleep(1);
    printf("O");    sleep(1);
    printf("\n");   sleep(1);
}
```

```
setvbuf(stdout, NULL, _IONBF, 0);
```

```
setvbuf(stdout, buf, _IOLBF, sizeof(buf));
```

```
setvbuf(stdout, buf, _IOFBF, sizeof(buf));
```

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

Buffered I/O – summary

VERY IMPORTANT

- Now, you know the buffer is just a piece of memory.
  - So, you need to be careful when you are playing with **`fork()`** and **`pthread_create()`**.

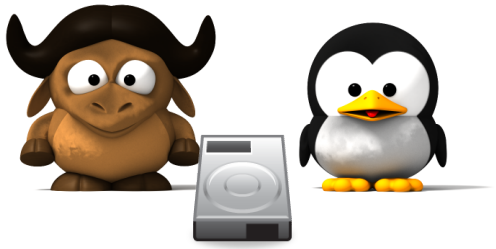
- **Challenge.** What will be the output?

```
int main(void) {
    printf("Hello");
    fork();
    printf("\n");
    return 0;
}
```

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

Looking at FS from the userspace

- GNU C Library call VS System call?
  - Buffered I/O and efficiency.
  - What is the true meaning of EOF?



Library call VS System call – what is EOF?

- Well, the following is just one of the common usage of EOF:

```
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);
}
```

Do you know what EOF really is?

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

Library call VS System call – what is EOF?

- First of all, **you can't find any "*EOF character*"** when using system calls.

```
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);
}
```

No more bytes to read.

Any "**WoW!**"?

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

Library call VS System call – what is EOF?

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

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

- That means: all those "**`f*()`**" functions *memorize* whether the end of file is reached or not!
  - If yes, it just returns -1 (EOF)!
  - If no, it either reads data from the buffer or system calls.

- **Main point: No EOF character in any files!**

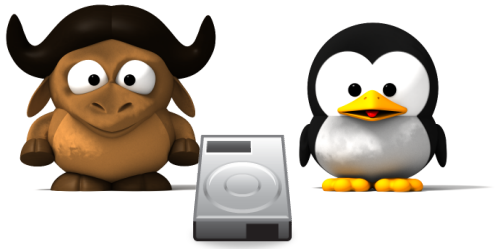
Summary

- The GNU I/O library functions give you a lot of convenience:
  - Great functions: **`fscanf()`**, **`fprintf()`**

- Yet, this abstracts away many truths and introduces (not on purpose) mis-conceptions!
  - From now on, you should never said: "***An empty file has an EOF character at the end of the file!***"
  - Also, use ***`feof()`*** with great care!

Looking at FS from the userspace

- GNU C Library call VS System call?
  - Buffered I/O and efficiency.
  - What is the true meaning of EOF?
- File and directory.
  - basics;



Attributes

- First, both files and directories are very similar:
  - They **both** have two kinds of data: **attributes** and **data**.
  - Attributes are as important as data:
    - Can you read the data correctly *without the size attribute*?

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
# _
```

File size

File type

Writing attributes?

- Can you change those attributes directly?

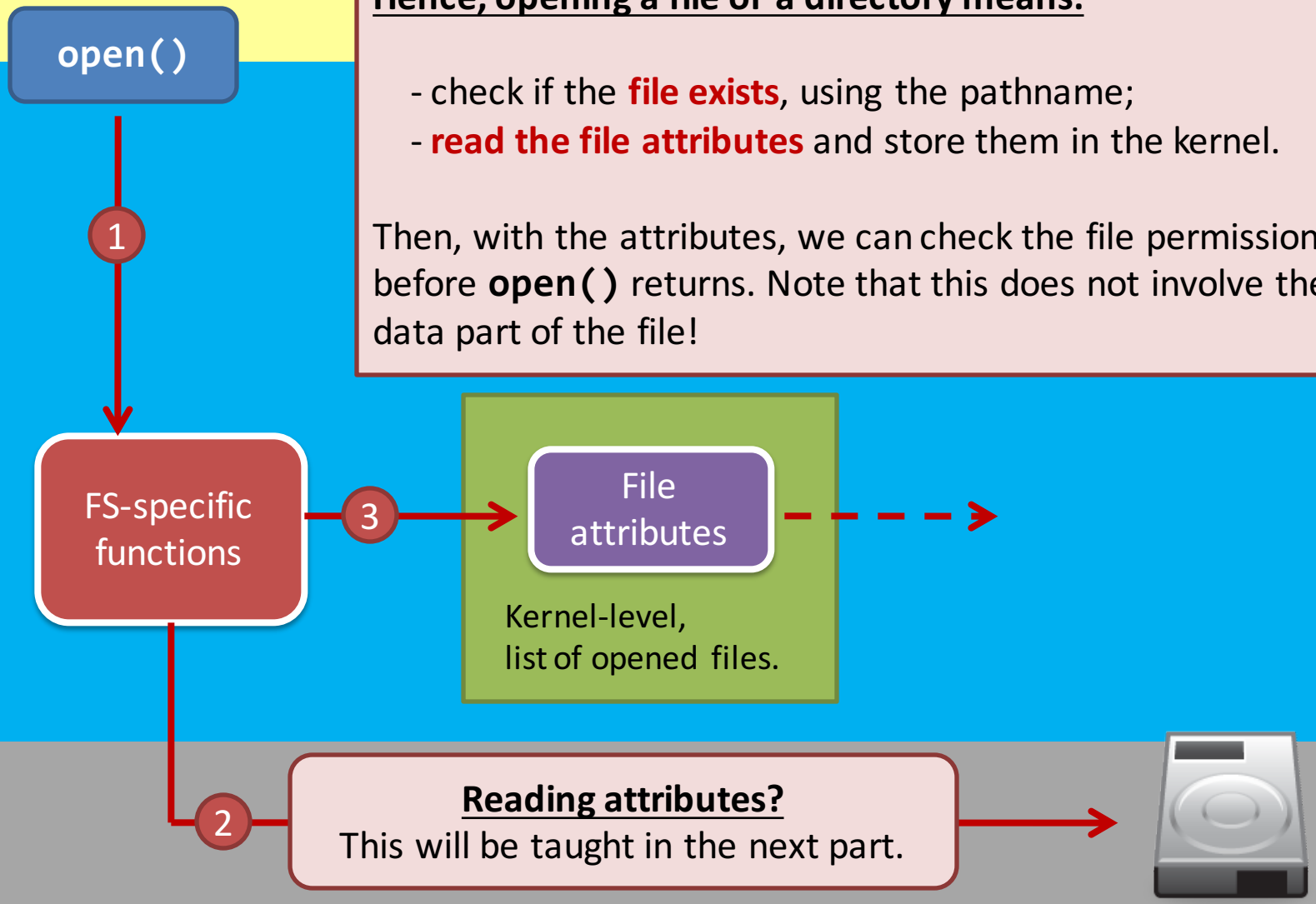
Common Attributes	Way to change them?	
	Command?	Syscall?
Name	`mv`	`rename()`
Size	Too many tools to update files' contents	`write()`, `truncate()`, etc.
Permission	`chmod`	`chmod()`
Owner	`chown`	`chown()`
Access, creation, modification time	`touch`	`utime()`

Still remember the flow of **open()**?

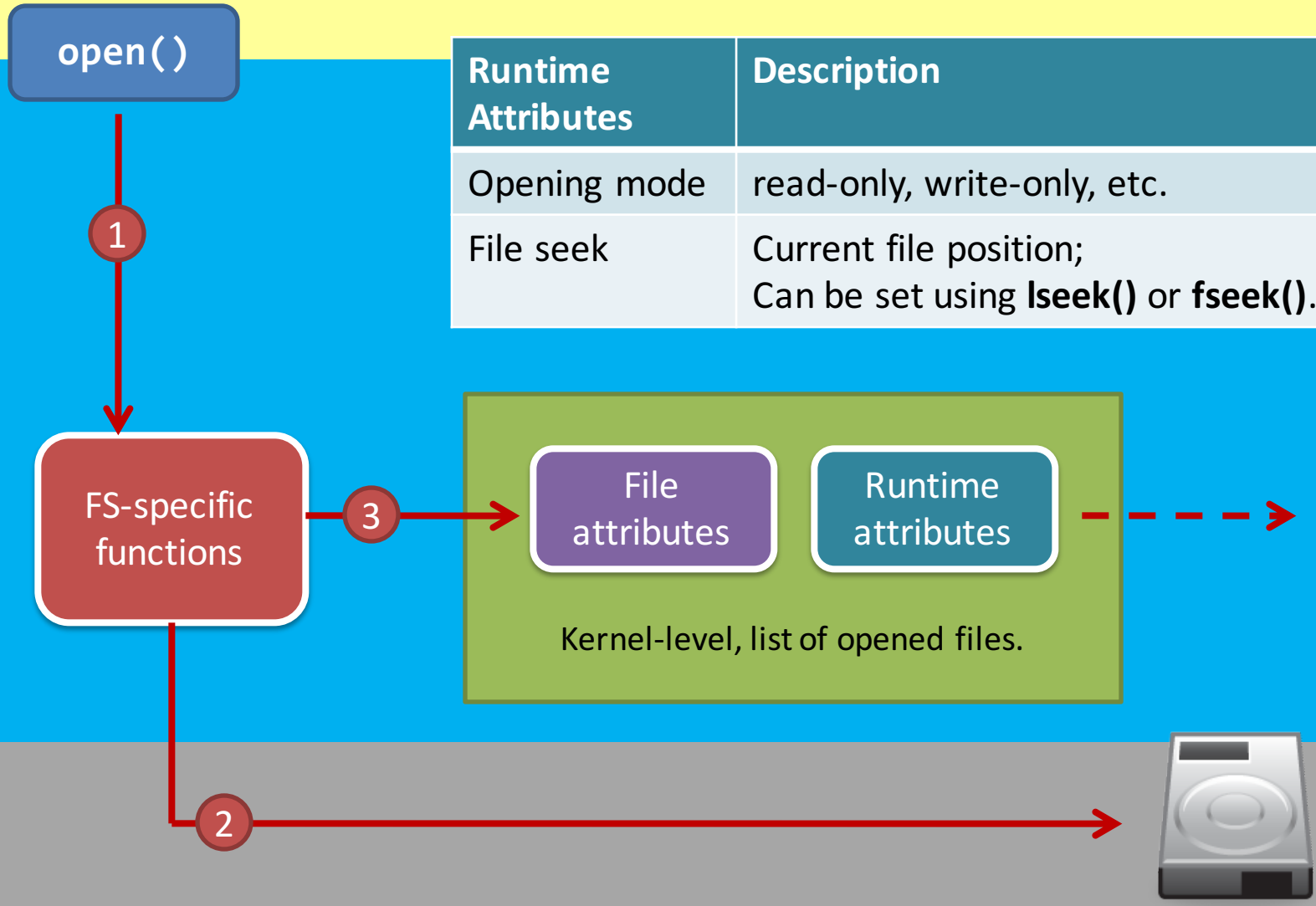
Hence, opening a file or a directory means:

- check if the **file exists**, using the pathname;
- **read the file attributes** and store them in the kernel.

Then, with the attributes, we can check the file permission before **open()** returns. Note that this does not involve the data part of the file!

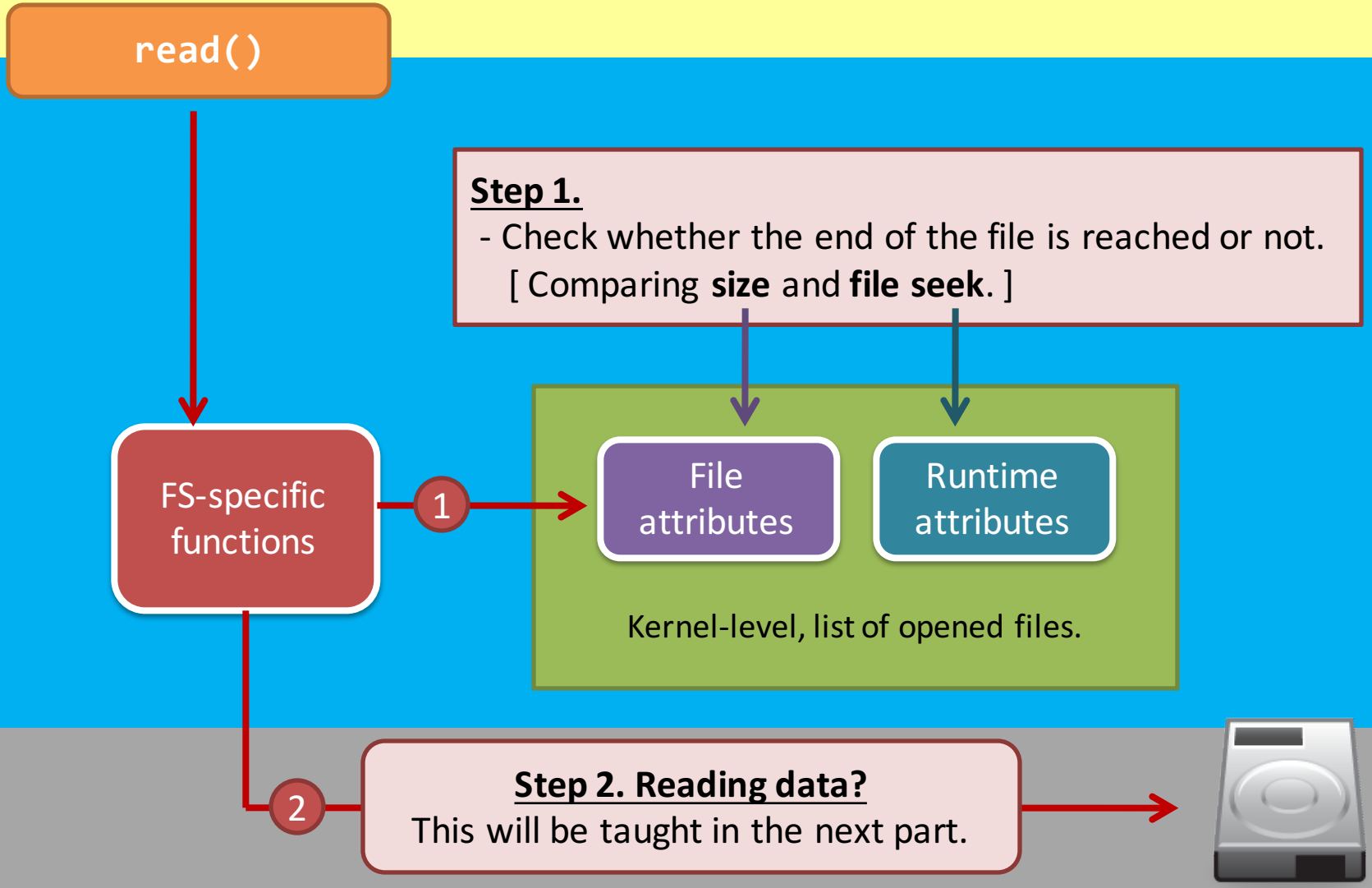


Still remember the flow of **open()**?

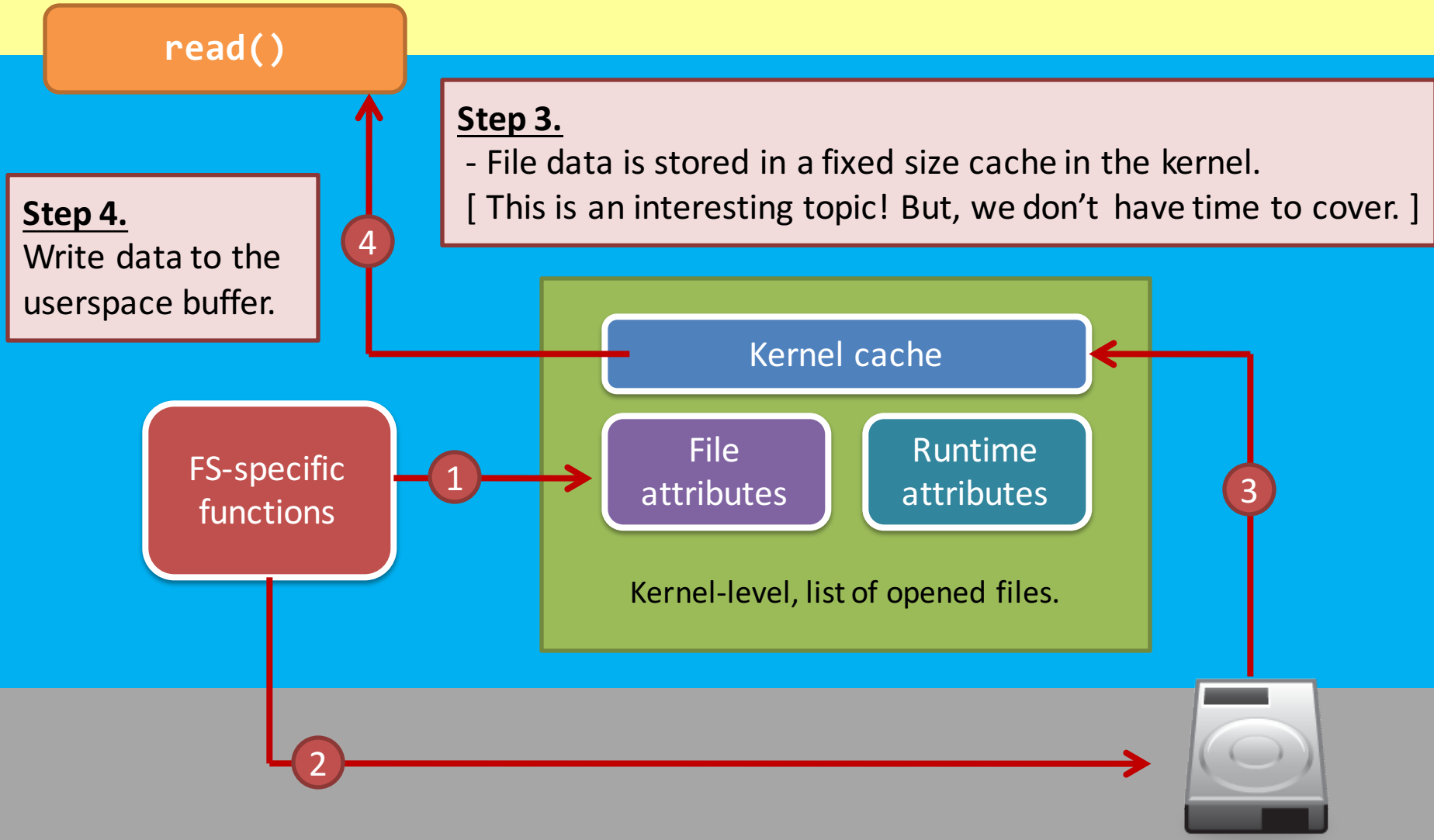


Runtime Attributes	Description
Opening mode	read-only, write-only, etc.
File seek	Current file position; Can be set using **lseek()** or **fseek()**.

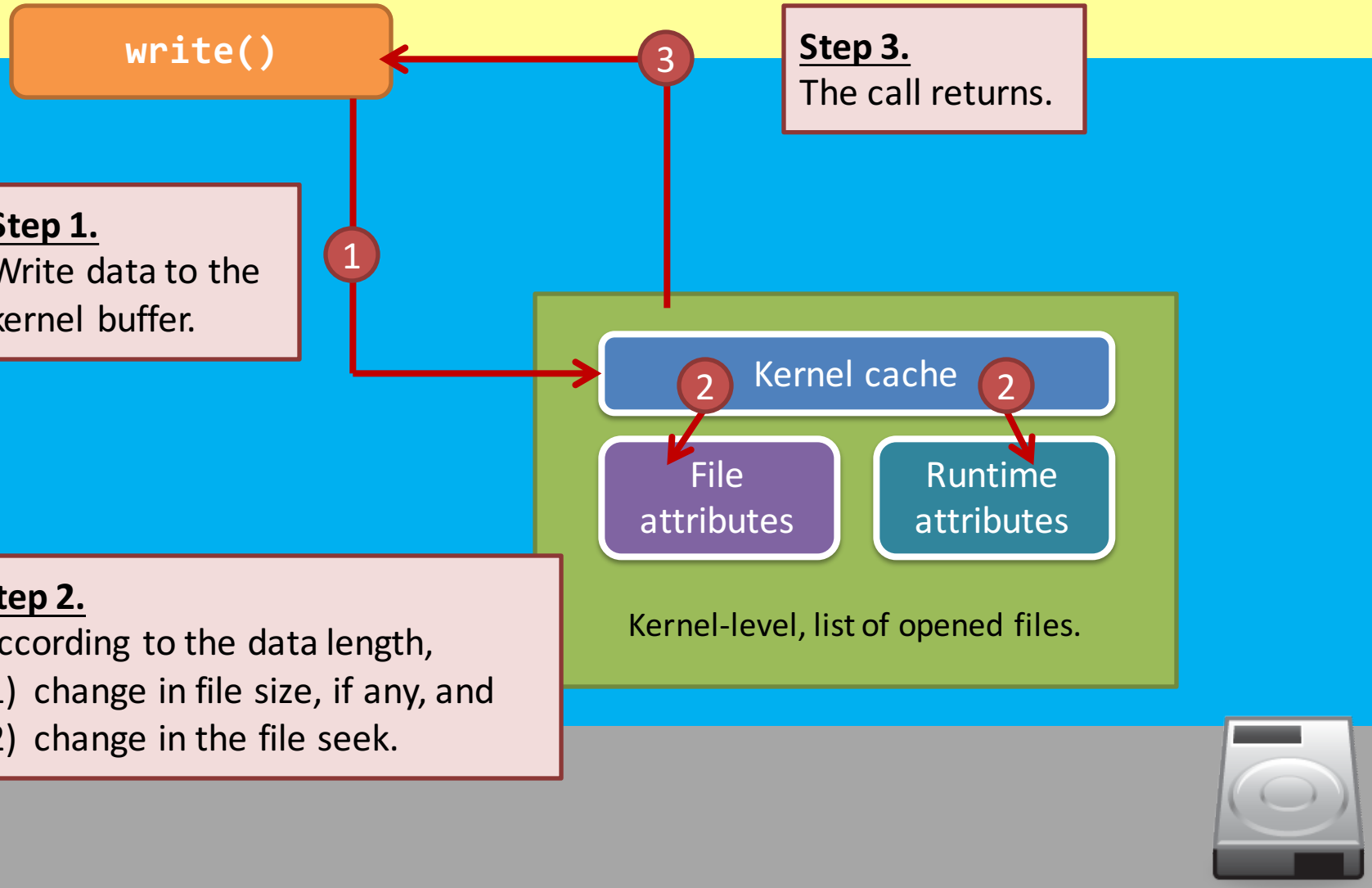
How about the `read()` system call?



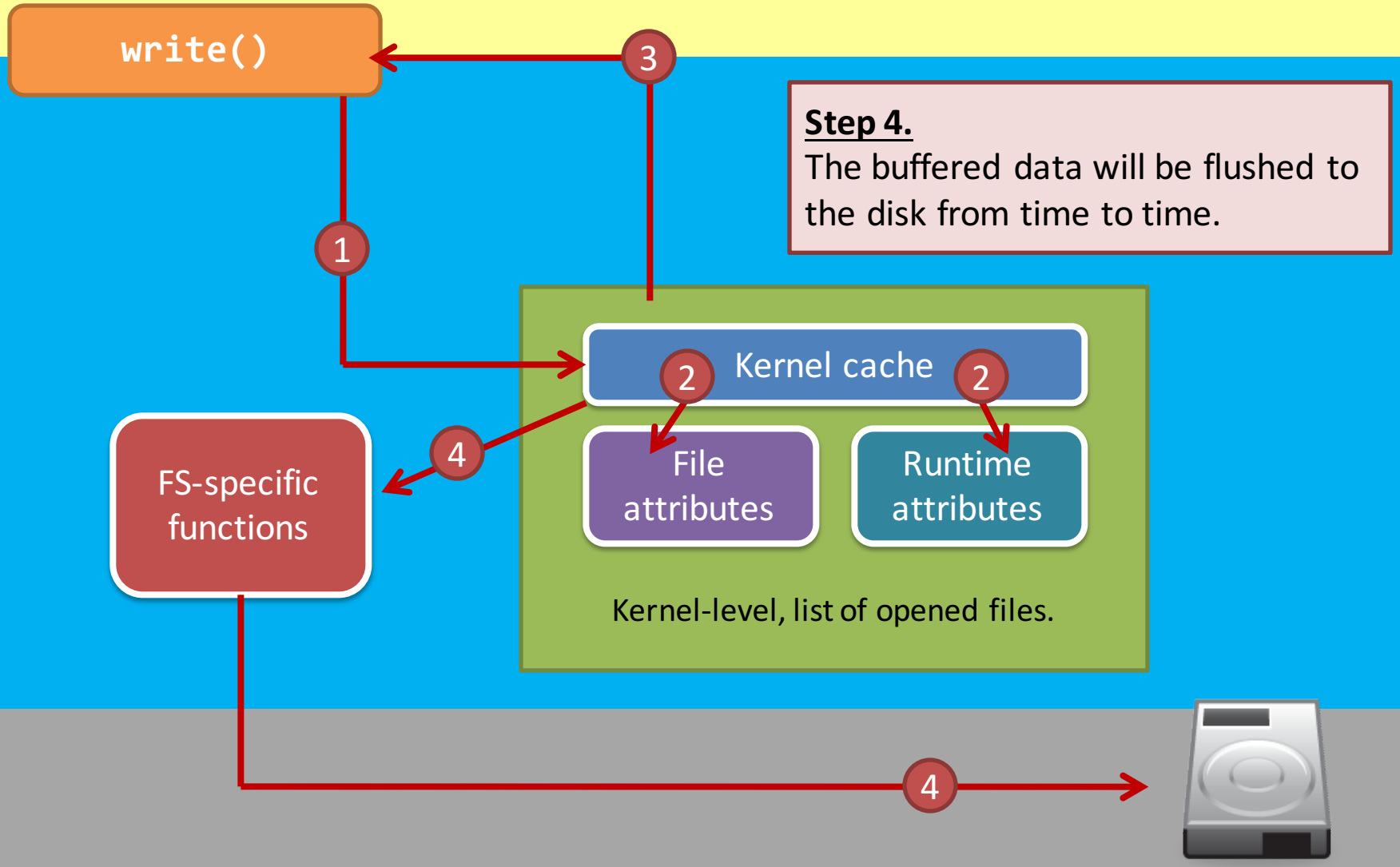
How about the `read()` system call?



How about the `write()` system call?



How about the `write()` system call?

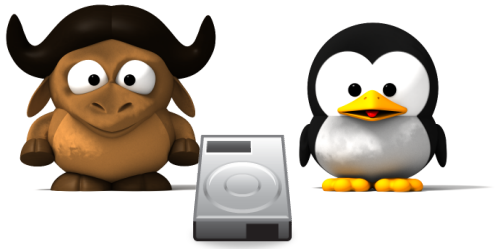


The kernel buffer cache implies…

- Class Discussion!
  - Increase reading performance?
  - Increase writing performance?
  - Can you answer me why **you cannot press the reset button**?
  - Can you answer me **why you need to press the "eject" button before removing USB drives?**

Looking at FS from the userspace

- GNU C Library call VS System call?
  - Buffered I/O and efficiency.
  - What is the true meaning of EOF?
- File and directory.
  - basics;
  - playing with directories.



Reading a directory

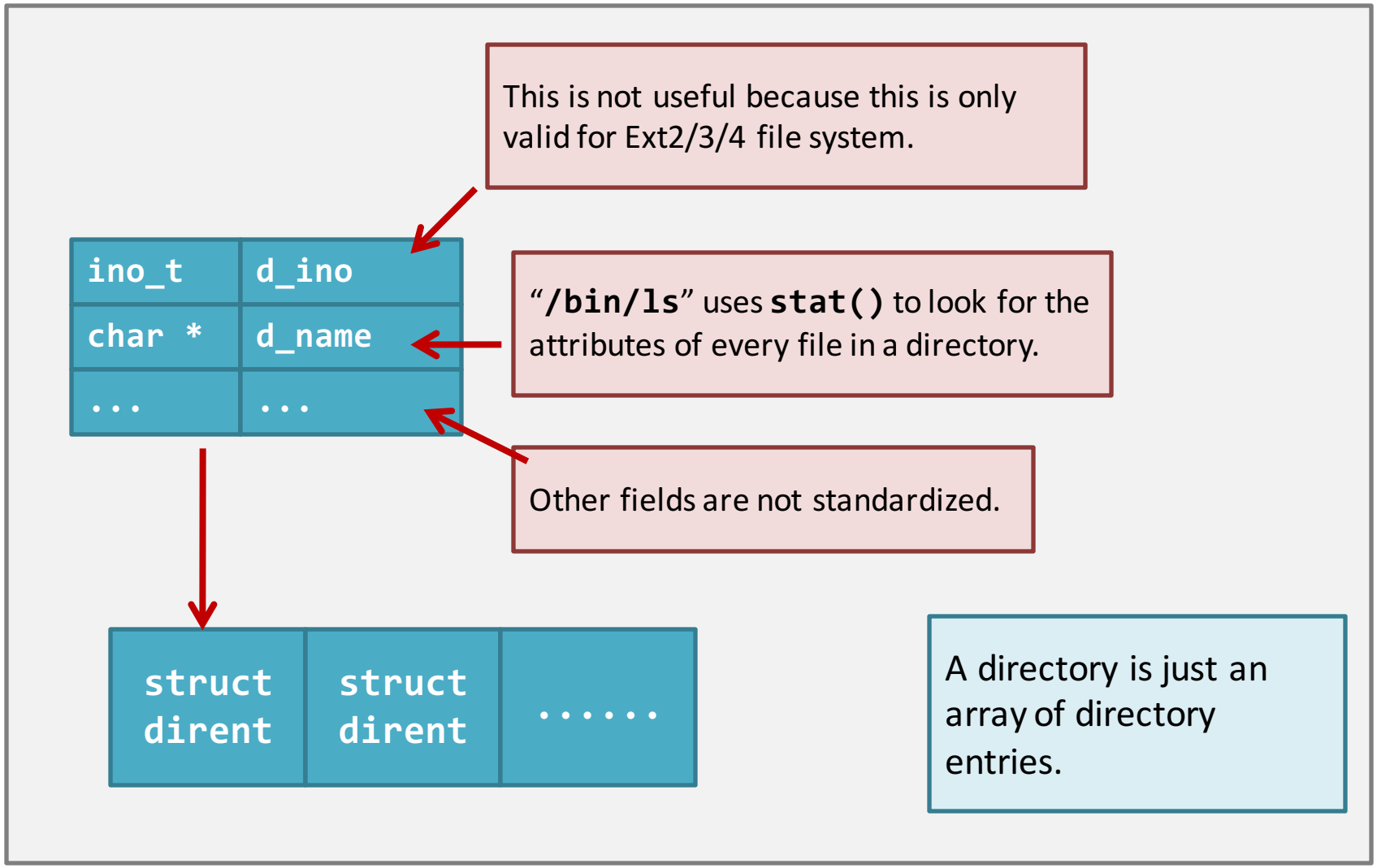
- Similarly, reading a directory also involves attributes and data.
  - Yet, you don't know how to play with in the userspace.

```
int main(int argc, char **argv) {
    DIR *dir;
    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
```

Directory entries – read



Directory entries – write

Removing an existing file also means writing to a directory. But, the write operation is to erase something…

The truth of file deletion: the system **will not** remove the corresponding directory entry in hosting directory completely.

[What evil things can you think of?]

Add a new file to a directory means appending data to the directory file.

This is a directory file: it contains an array of directory entries.

Directory entries – write

- Example screenshot.

```
$ ls -ld .
drwxr-xr-x 11 tywong tywong 4096 2010-11-16 22:20 .

$ ls -l final.xls
ls: cannot access final.xls: No such file or directory

$ touch final.xls

$ ls -l final.xls
-rw-r--r-- 1 tywong tywong 0 2010-11-21 16:05 final.xls

$ ls -ld .
drwxr-xr-x 11 tywong tywong 4096 2010-11-21 16:05 .

$ rm final.xls
rm: remove regular empty file `final.xls'? y

$ ls -ld
drwxr-xr-x 11 tywong tywong 4096 2010-11-21 16:06 .
```

Create a file updates the directory file.

Remove a file also updates the directory file.

Summary

- Through this part, we learn:
  - the truth about the calls that we usually use,
  - the content of a file is not the only entity, but also the file attributes.

- In the next part, we will go into the disk:
  - How and where to store the file attributes?
  - How and where to store the data?
  - How to manage a disk?