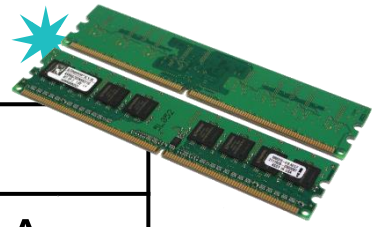
CSE251: System Programming

24. System-Level I/O

Seongil Wi

Department of Computer Science and Engineering

Today's Topic



Control
(brain)

01010010

Datapath

Processor
(CPU)
(e.g., Intel I7)

Data A

01010010

00110110

Main memory

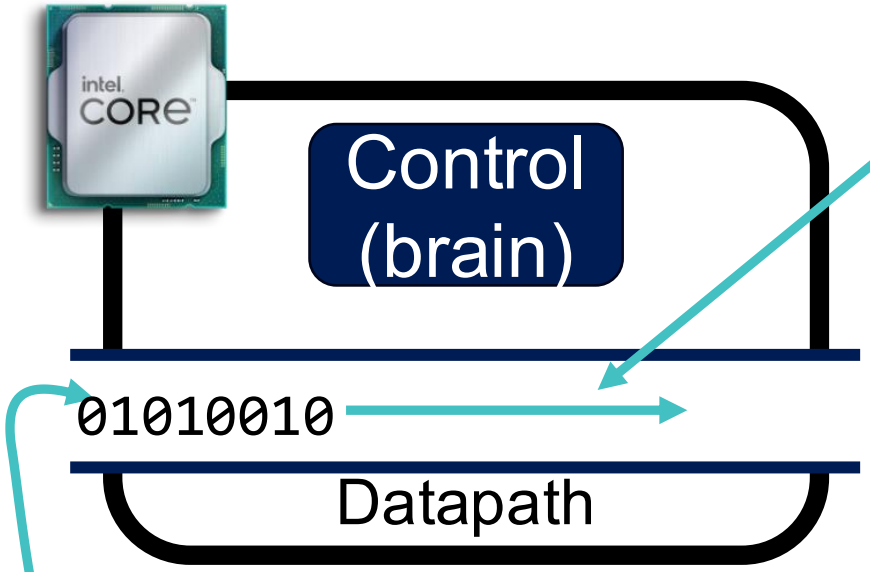
01010010
00110110
Program A

11001110
01000110
Program B

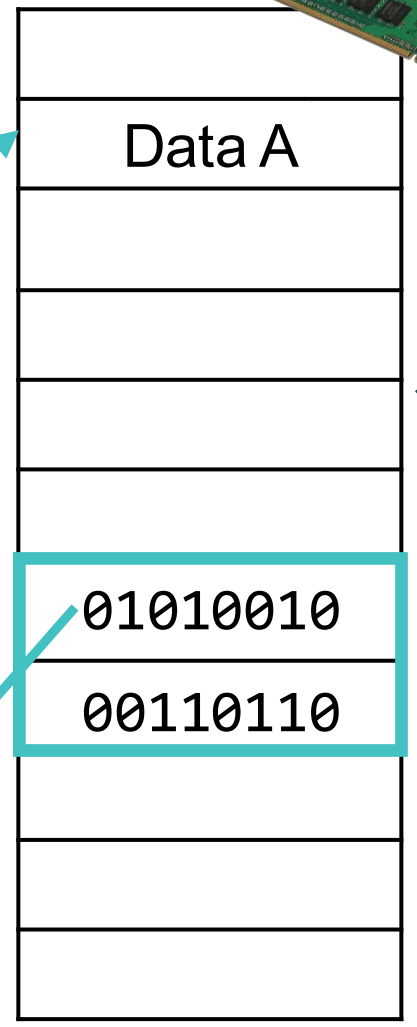
01101101
11010111
Program C

Hard disk

Today's Topic



Processor
(CPU)
(e.g., Intel I7)



Main memory

External
devices

Today's topic:
System-level I/O

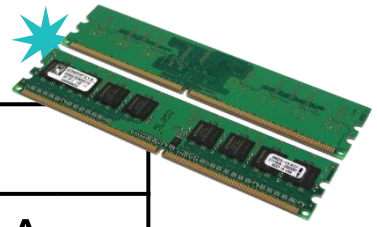
Today's Topic

Control (brain)

01010010

Datapath

Processor (CPU) (e.g., Intel I7)



Data A

01010010

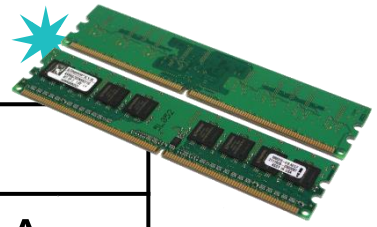
00110110

Main memory

External devices

System output

Today's Topic



Control (brain)

01010010

Datapath

Processor (CPU) (e.g., Intel I7)

Data A

01010010

00110110

Main memory

External devices

System input

I/O (Input/Output)

System-Level I/O

- The communication between main memory and external device

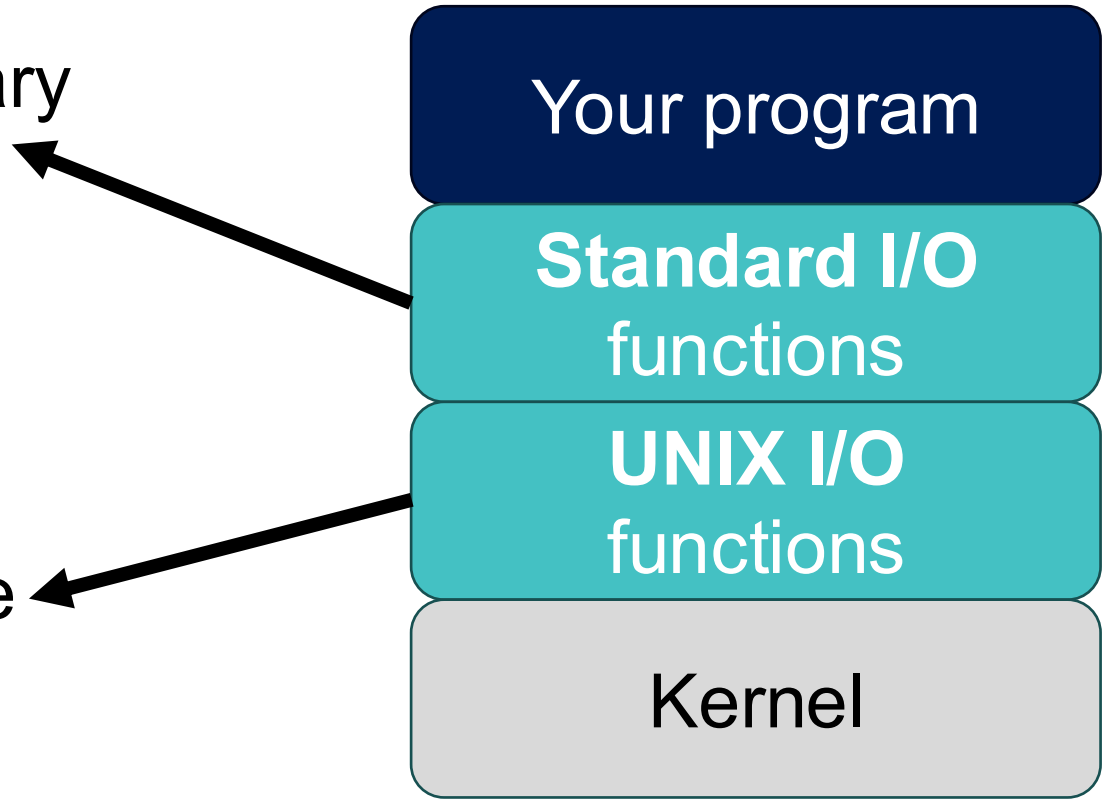
HDD, SSD, Network,
Monitor, Keyboard, …

System-Level I/O

- The communication between main memory and external device

- **Standard I/O:** a high-level C library interface for input and output
  - e.g., `printf`

- **Unix I/O:** a low-level OS interface for input and output
  - e.g., `write`



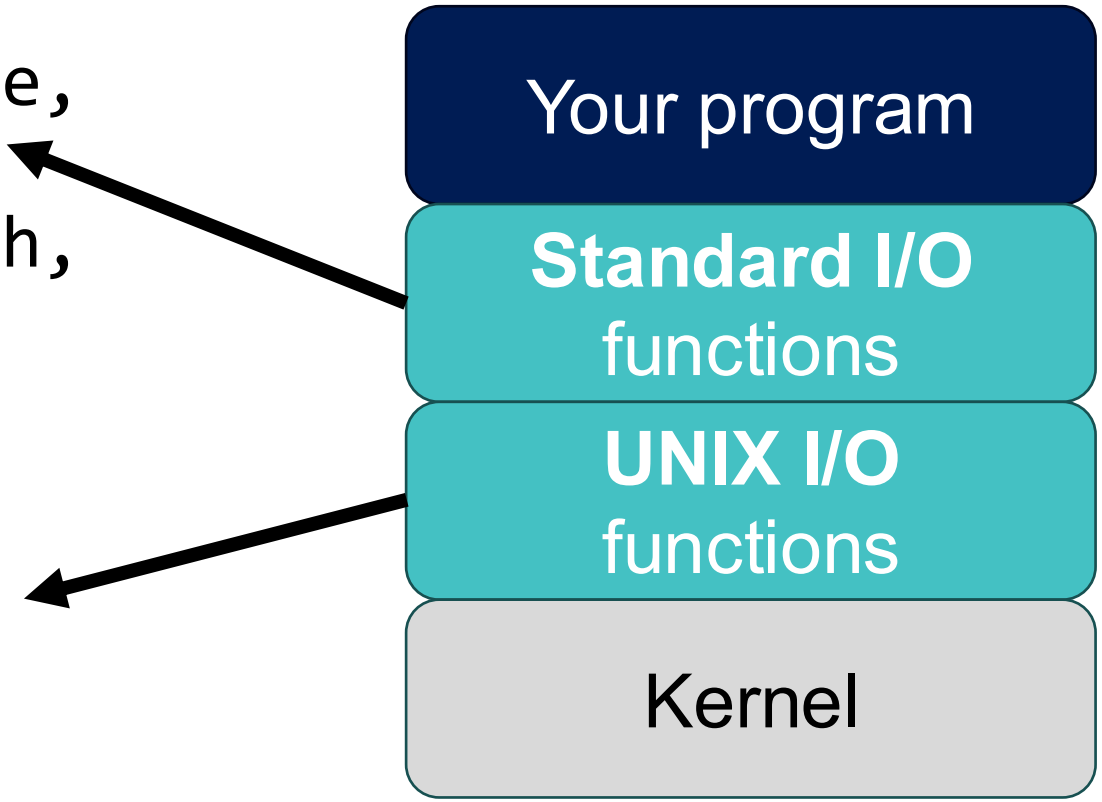
System-Level I/O

- The communication between main memory and external device

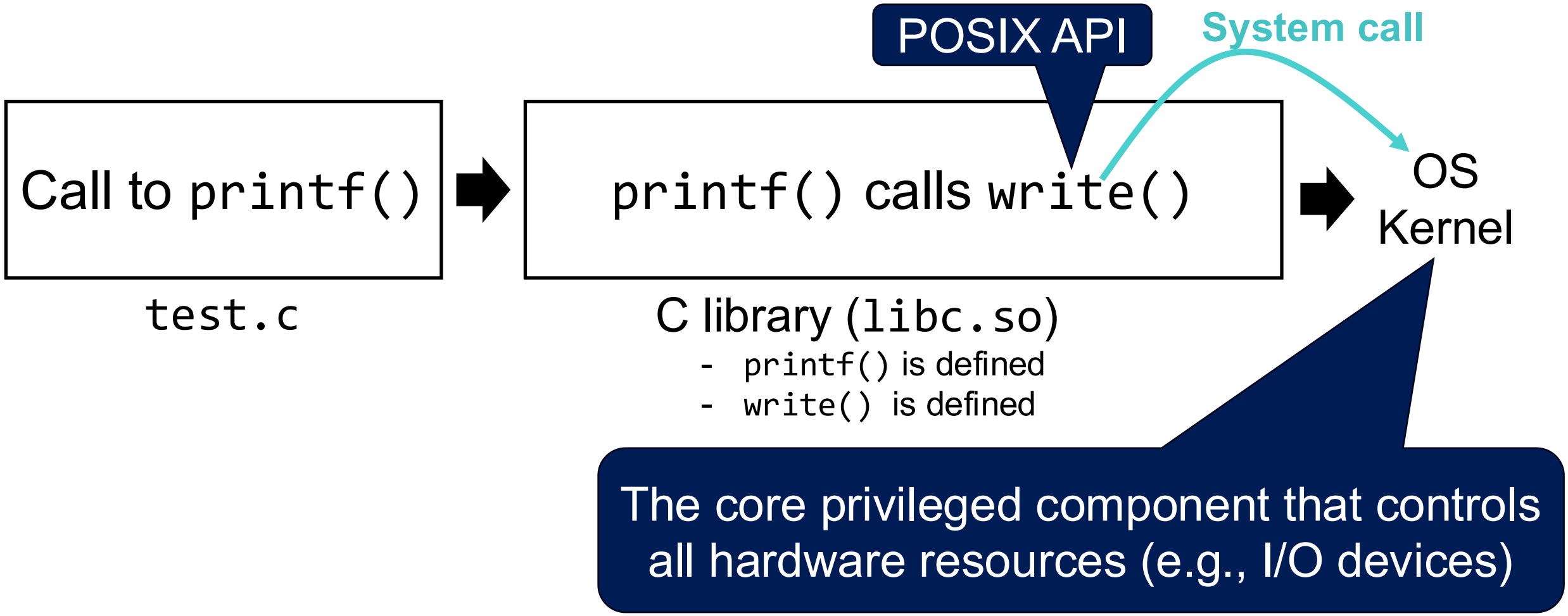
- `fopen, fdopen, fread, fwrite, fscanf, fprintf, sscanf, sprint, fgets, fputs, fflush, fseek, fclose`

Call

- `open, read, write, lseek, stat, close`



(Recap) Library Calls Workflow



Unix I/O

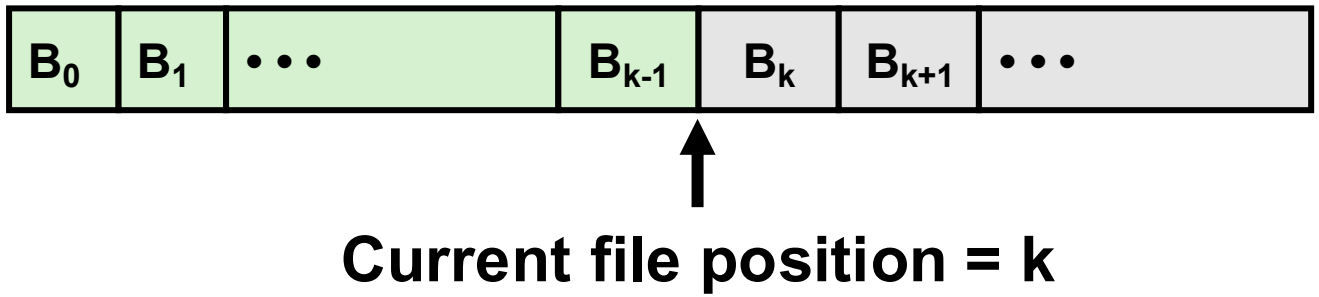
Unix I/O Overview

- A Linux ***file*** is a sequence of m bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

- **Almost everything is a file in Linux**
  - All I/O devices are represented as files:
    - `/dev/sda2` (e.g., disk partition)
    - `/dev/tty2` (terminal)
    - `/dev/null` (discard all writes / read empty file)
  - The kernel is represented as a file:
    - `/boot/vmlinuz-3.13.0-55-generic` (kernel image)
    - `/proc` (pseudo files to access kernel data structures)

Unix I/O Overview

- **Kernel offers a set of basic operations for all files**
  - Opening and closing files
    - `open()` and `close()`
  - Reading and writing a file
    - `read()` and `write()`
  - Look up information about a file (size, type, last modification time, …)
    - `stat()`, `lstat()`, `fstat()`
  - Changing the *current file position* (seek)
    - indicates next offset into file to read or write
    - `lseek()`



File Types

- **Each file has a *type* indicating its role in the system**
  - Regular file: Stores arbitrary data
  - Directory: Index for a related group of files
  - Socket: For communicating with a process on another machine

- **Other file types beyond our scope**
  - Named pipes (FIFOs)
  - Symbolic links
  - Character and block devices (e.g., Keyboard)

Regular Files

- **A regular file contains arbitrary data**
- **Applications often distinguish between text and binary files**
  - Text files contain human-readable text
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn't care! It's all just bytes!
- **Text file is sequence of text lines**
  - Text line is sequence of chars terminated by newline char (**'\n'**)
- **End of line (EOL) indicators in other system:**
  - Linux and MacOS: **'\n'** (0xa)
    - Line feed (LF)
  - Windows: **'\r\n'** (0xd 0xa)
    - Carriage return (CR) followed by line feed (LF)

Directories

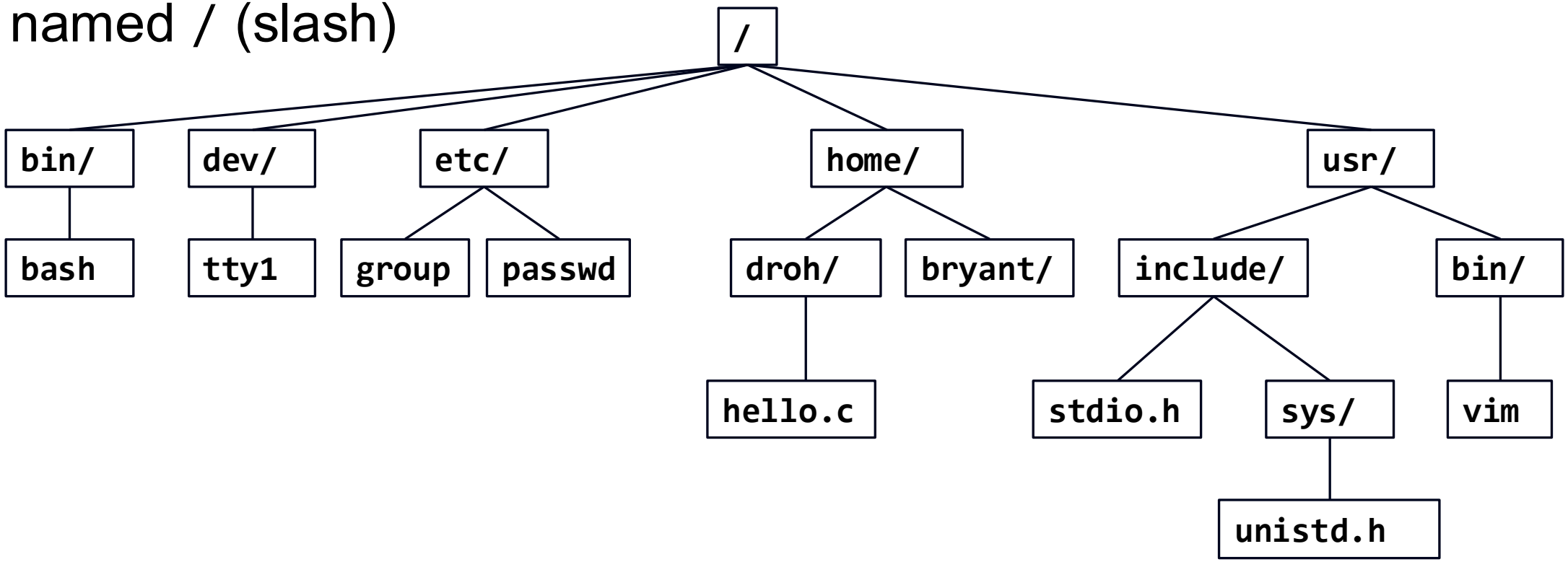
- **Directory consists of an array of entries** (also called *links*)
  - Each entry maps a filename to a file

- **Each directory contains at least two entries**
  - `.` (dot) maps to the directory itself
  - `..` (dot dot) maps to the parent directory in the *directory hierarchy* (next slide)

- **Commands for manipulating directories**
  - **`mkdir`**: create empty directory
  - **`ls`**: view directory contents
  - **`rmdir`**: delete empty directory

Directory Hierarchy

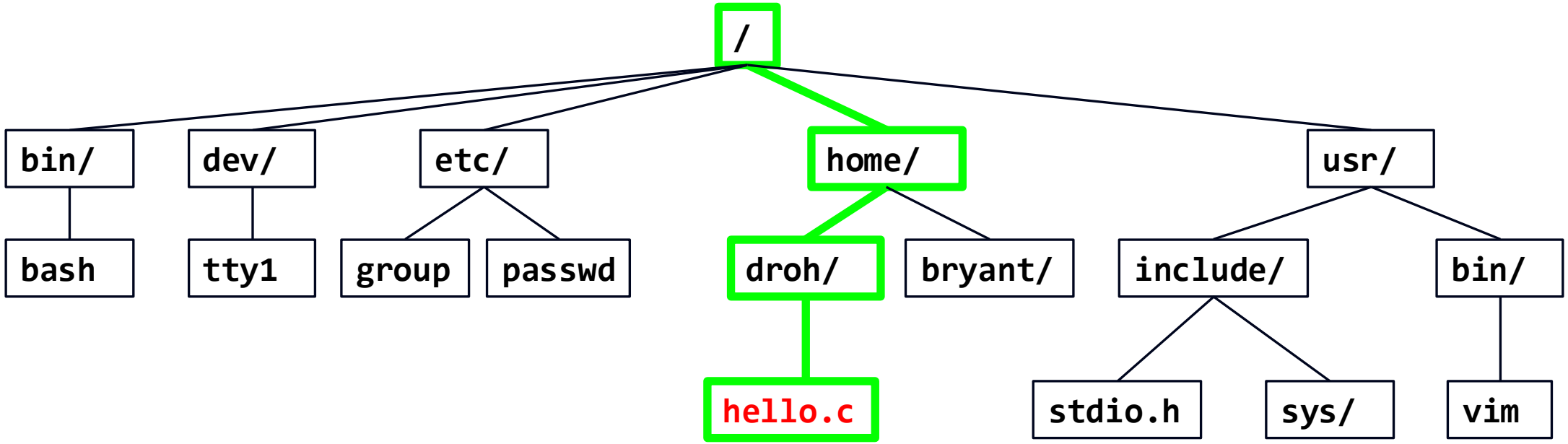
- All files are organized as a hierarchy anchored by root directory named / (slash)



- **Kernel maintains current working directory (*cwd*) for each process**
  - Modified using the cd command

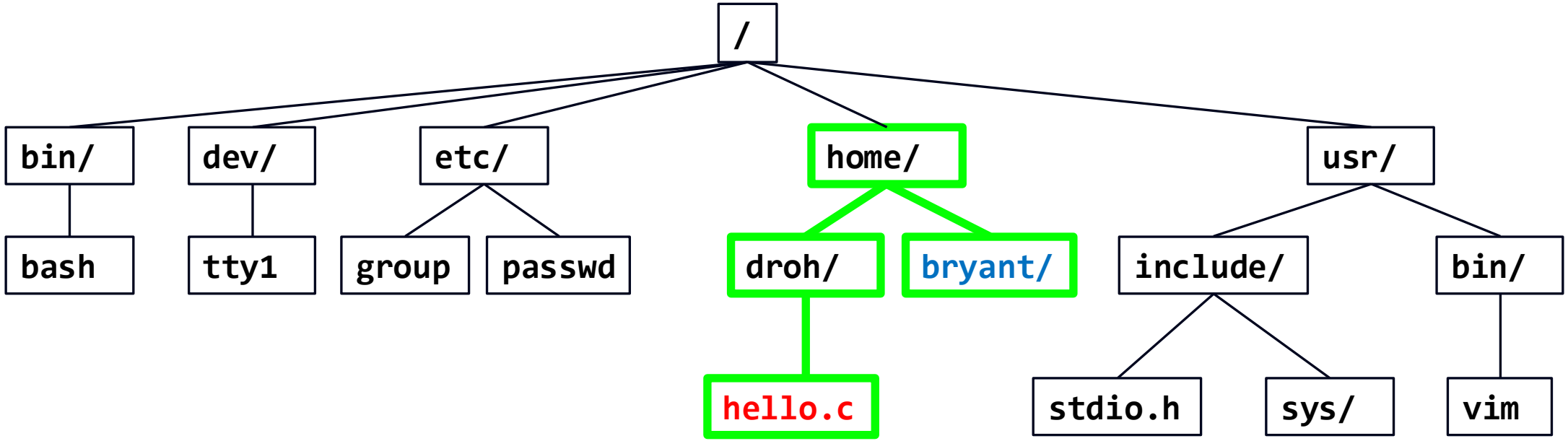
Pathnames

- **Locations of files in the hierarchy denoted by pathnames**
  - Absolute pathname starts with '/' and denotes path from root
    - `/home/droh/hello.c`



Pathnames

- **Locations of files in the hierarchy denoted by pathnames**
  - Absolute pathname starts with '/' and denotes path from root
    - `/home/droh/hello.c`
  - Relative pathname denotes path from current working directory (cwd)
    - `../droh/hello.c` (cwd: `/home/bryant`)



Opening Files

- Opening a file **informs the kernel** that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer ***file descriptor***
  - fd == -1 indicates that an error occurred

File Descriptor (FD)

- A **small integer** used to access an open file, device, pip, or socket

- Each process begins life with three open files (managed by “File Descriptor Table”)
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

- Since FDs 0, 1, and 2 are usually already open (reserved), the first open() normally returns FD 3

CF. Lots of ways to call open

Open an existing file:

`open(path, flags)`

`flags` must include exactly one of:

`O_RDONLY`	Only want to read from file
`O_WRONLY`	Only want to write to file
`O_RDWR`	Want to do both

Flags may also include (use | to combine)

`O_APPEND`	All writes go to the very end
`O_TRUNC`	Delete existing contents if any
`O_CLOEXEC`	Close this file if execve() is called

Open or create a file:

`open(path, flags, mode)`

`flags` must include

`O_CREAT`	Create the file if it doesn't exist

and exactly one of:

`O_WRONLY`	Only want to write to file
`O_RDWR`	Want to write and read

and maybe also some of:

`O_EXCL`	Fail if file does exist
`O_APPEND`	All writes go to the very end
`O_TRUNC`	Delete existing contents if any
`O_CLOEXEC`	Close this file if execve() is called

(and many more… consult the open() manpage)

Do not memorize!

Closing Files

- Closing a file **informs the kernel** that you are finished accessing that file

```
if (close(fd) < 0) {
    fprintf(stderr, "%s: write error: %s",
            filename, strerror(errno));
    exit(1);
}
```

- Closing an already closed file descriptor is dangerous…
  - … because the same descriptor number may later be reused
  - … you may accidentally close the file instead

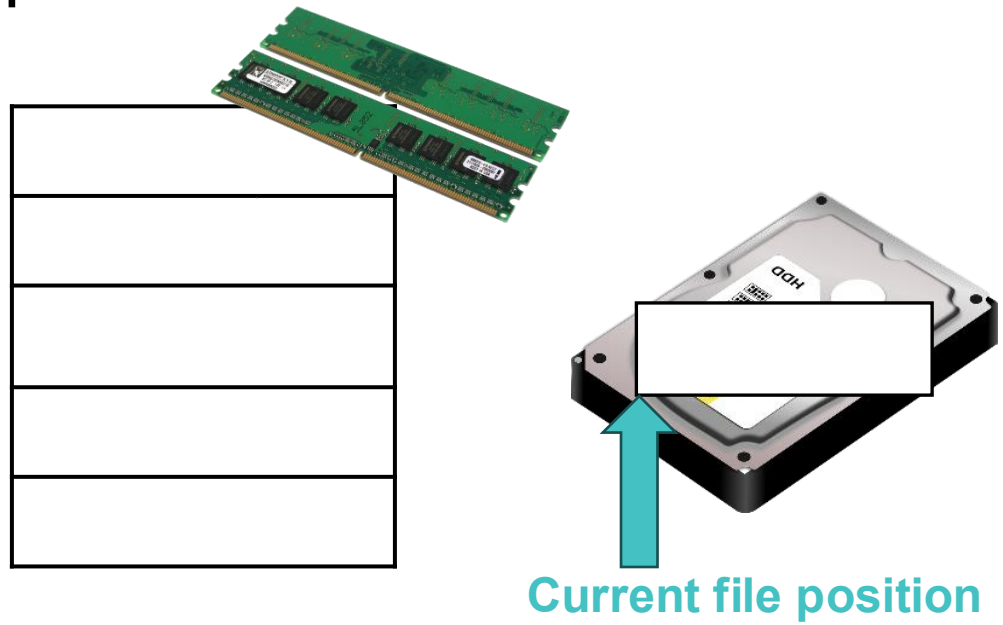
  - Before **`close()`**, always check if **`fd`** is valid (i.e., **`fd != -1`**)
  - After **`close()`**, assign an invalid **`fd`** (i.e., **`fd = -1`**)

Reading Files

- Reading a file **copies bytes from the current file position to memory**, and then updates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

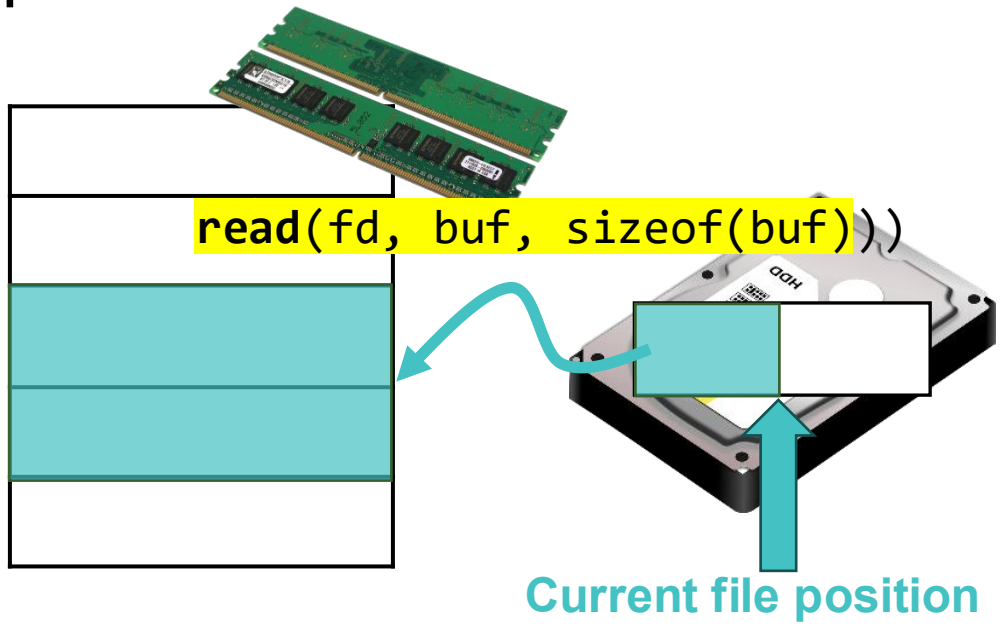


Reading Files

- Reading a file **copies bytes from the current file position to memory**, and then updates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

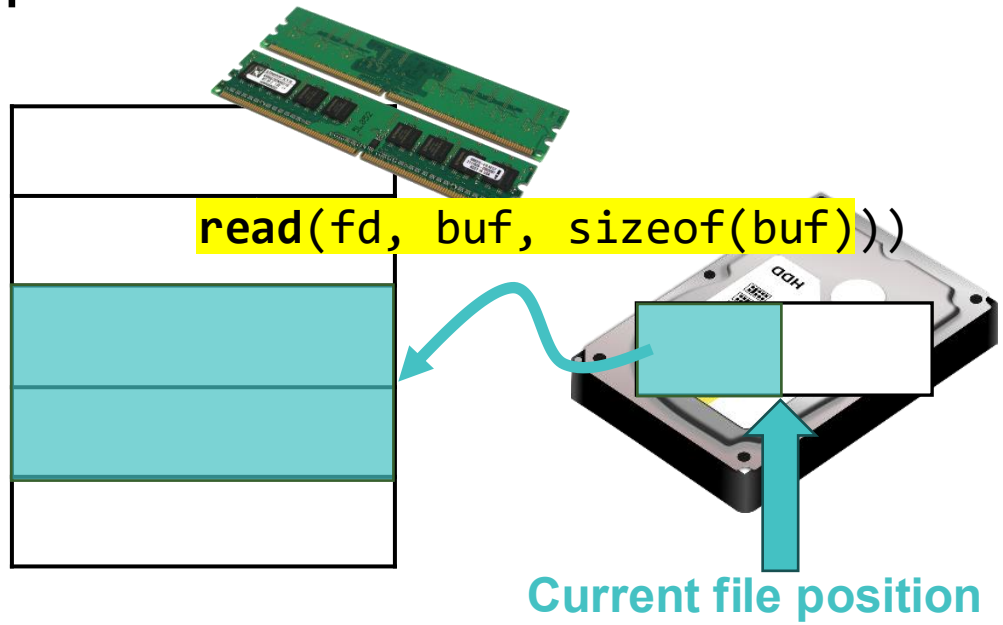


Reading Files

- Reading a file **copies bytes from the current file position to memory**, and then updates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```



- Returns number of bytes read from file **`fd`** into **`buf`**
  - Return type **`ssize_t`** is signed integer (cf, **`size_t`** is unsigned integer)
  - **`nbytes < 0`** indicates that an error occurred
  - Short counts (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Writing Files

- Writing a file **copies bytes from memory to the current file position**, and then updates current file position

```
char buf[512];
int fd;        /* file descriptor */
int nbytes;    /* number of bytes written */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from **`buf`** to file **`fd`**
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

On Short Counts

- Processes **fewer bytes than requested**
  - e.g., `read(fd, buf, 100)` requests 100 bytes, but returns 40 because only 40 bytes were currently available

- Short counts can occur in these situations:
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets, pipes, etc.

- Best practice is to always allow for short counts

Simple Unix I/O Example

- Copying stdin to stdout, one byte at a time

```
#include <unistd.h>

int main(void) {
    char c;

    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);

    return 0;
}
```

Any problems?

`read()` and **`write()`** are system calls, so they are expensive. Many small I/O operations can cause significant overhead

Home-grown Buffered I/O Code

- Copying file to stdout, BUFSIZE bytes at a time

```
#define BUFSIZE 64

int main(void) {
    char buf[BUFSIZE];

    while((nread = read(infd, buf, BUFSIZE)) != 0)
        write(STDOUT_FILENO, buf, nread);
    return 0;
}
```

For time efficiency, it is recommended to read or write data in large chunks

Standard I/O

Standard I/O Functions

- The C standard library (libc.so) contains a collection of higher-level ***standard I/O*** functions
  - https://man7.org/linux/man-pages/dir_section_3.html

- **Examples of standard I/O functions:**
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

Standard I/O Streams

- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory

- C programs begin life with three open streams (defined in **`stdio.h`**)
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Buffered I/O: Motivation

- **Applications often read/write one character at a time**
  - `getc`, `putc`, `ungetc`
  - `gets`, `fgets`
    - Read line of text one character at a time, stopping at newline
- **Motivation**: Implementing as Unix I/O calls is expensive
  - `read` and `write` require Unix kernel calls
    - System call takes more than 10,000 clock cycles

Recap: Unix I/O Call is Expensive!

- Copying stdin to stdout, one byte at a time

```
#include <unistd.h>

int main(void) {
    char c;

    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);

    return 0;
}
```

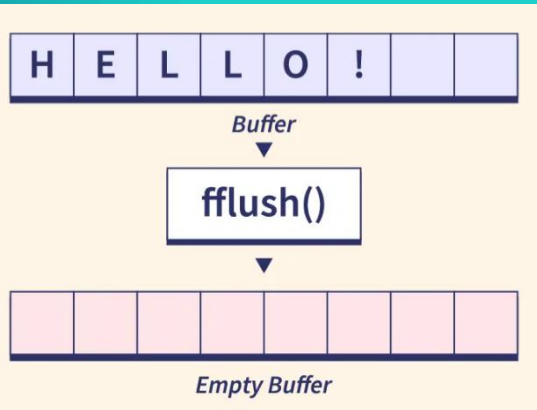
Any problems?

`read()` and **`write()`** are system calls, so they are expensive. Many small I/O operations can cause significant overhead

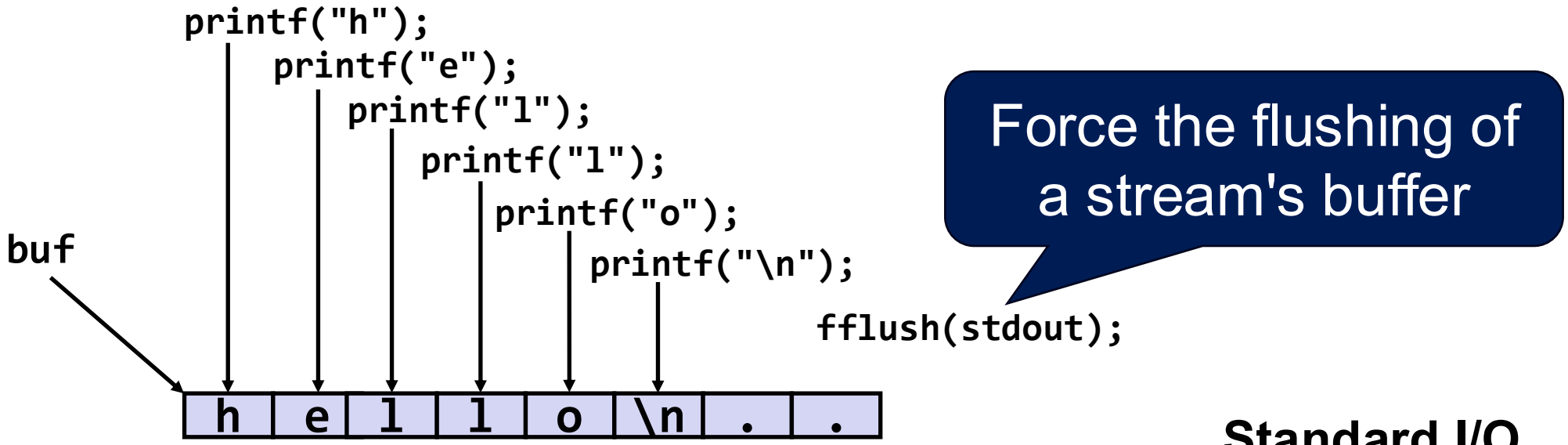
Buffered I/O: Motivation

- **Applications often read/write one character at a time**
  - `getc`, `putc`, `ungetc`
  - `gets`, `fgets`
    - Read line of text one character at a time, stopping at newline
- **Motivation**: Implementing as Unix I/O calls is expensive
  - `read` and `write` require Unix kernel calls
    - System call takes more than 10,000 clock cycles
- **Solution: Buffered read**
  - Use Unix `read` once to grab large block of bytes
  - User input functions take one byte at a time from buffer (**Fast!**)
    - Refill buffer when empty

Buffering in Standard I/O



- Standard I/O functions use **buffered I/O**



- Buffer flushed to output fd (monitor) on **"\n"**, call to **fflush** or **exit**, or return from **main**

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:
  - `strace` intercepts and records every system call made by a program

```
#include <stdio.h>

int main() {
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                  = 6
...
exit_group(0)                           = ?
```

Invoked only once

Unix I/O vs. Standard I/O

- **Standard I/O** is implemented using **Unix I/O**
- Which ones should you use in your programs?

Pros and Cons of Unix I/O

- **Pros**
  - Unix I/O is the most general form of I/O
    - All other I/O packages are implemented using Unix I/O functions
  - Unix I/O provides functions for accessing file metadata (using `stat()`)
  - Unix I/O functions can be used safely in signal handlers

- **Cons**
  - Dealing with short counts is tricky and error prone
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone

Pros and Cons of Standard I/O

- **Pros**
  - Buffering increases efficiency by decreasing the number of read and write system calls
  - Short counts are handled automatically

- **Cons**
  - Provides no function for accessing file metadata
  - Standard I/O functions are not appropriate for signal handlers
  - Standard I/O is not appropriate for input and output on network sockets

Question?