# C++ Classes & References

## Computer Systems Programming, Spring 2023

**Instructor:** Travis McGaha

**TAs:**

Kevin Bernat
Jialin Cai
Mati Davis
Donglun He
Chandravaran Kunjeti
Heyi Liu
Shufan Liu
Eddy Yang

# Logistics

- HW1 (FileReaders) Due Thursday 2/9 @ 11:59 pm
  - To be released shortly after Lecture
  - After this lecture, you should have everything you need to complete the assignment

# Pre-Semester Survey Response

- Expectations:
  - Learn C++
  - Learn systems stuff to flesh out what 5930 taught
  - Help with internships & future courses (5050, 5480, etc)
  - Multithreading, networking, etc.

- This course can’t do everything, trying to balance all of these while respecting the job hunt & other courses

# Pre-Semester Survey Response (Concerns)

- Struggle with C programming
  - C programming is required, but people usually find C++ to be “different” in a (usually) better way.
- 5930 is a pre-req
  - You don’t need everything from 5930. You just need:
    - A high-level idea of memory layout (stack & heap)
    - General understanding of what assembly is & how it works
    - C programming
- Difficulty
  - HW0 was on the harder side, future assignments should be better.

# Pre-Semester Survey Response (what works well)

- Things I already do:
  - Lecture recordings & posted slides
  - OH
  - Good visuals
  - Clear HW grading & specifications
  - Kindly answer "stupid" questions in lecture
  - Supporting/connecting assignments to lecture content
  - In class activities (I have had some, will try to add more)

- **Let me know if I can do any of these better though**

# Pre-Semester Survey Response (what works well)

- Things I sort of do?:
  - I don’t do pre-lecture quizzes & videos,
    but I do have recordings and quizzes afterwards.
  - Don’t have weekly "assignments", but I do have weekly check ins

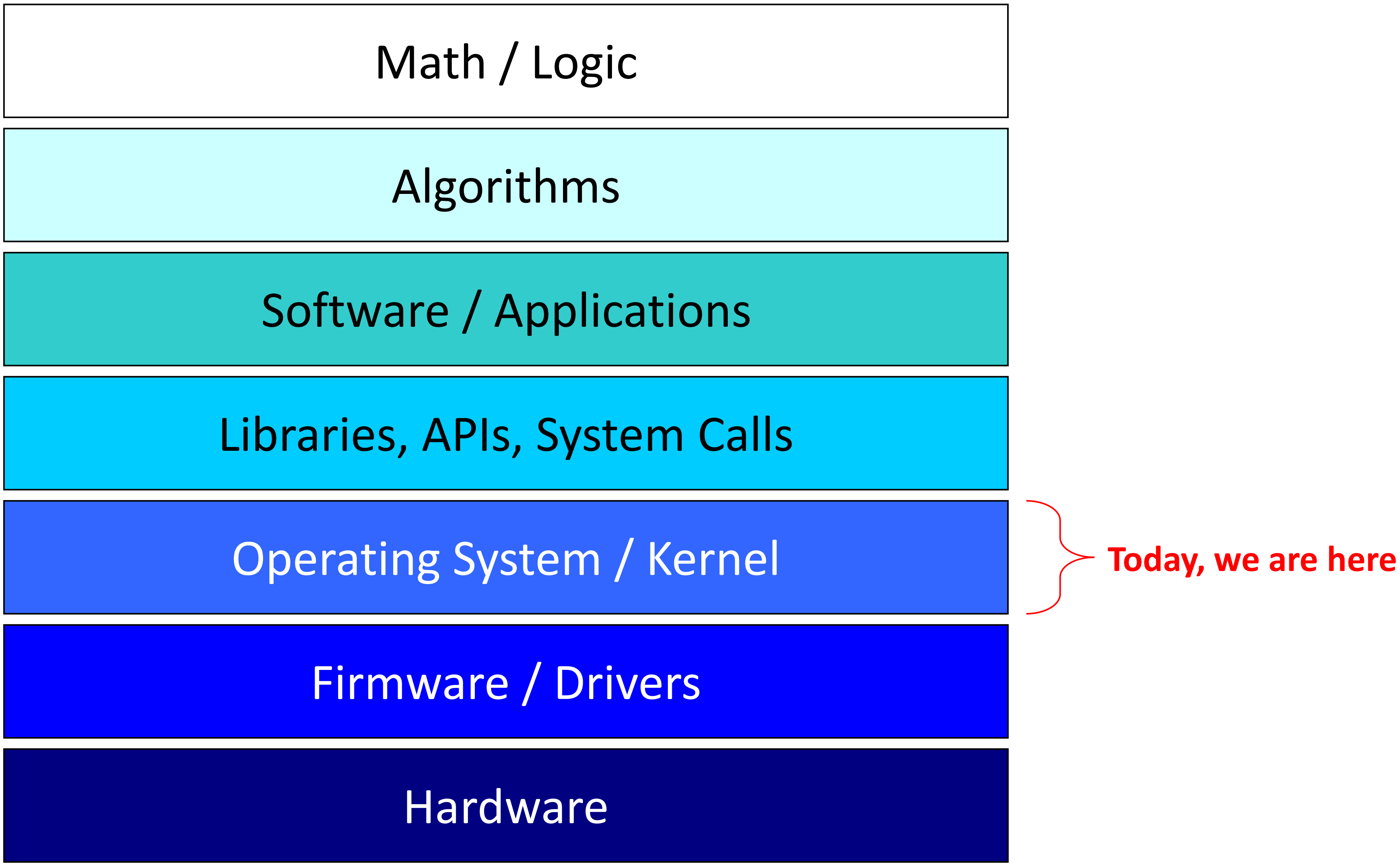
# Pre-Semester Survey Response (what works well)

- Things I do not do (as of now):
  - Exam guides & study guides
    - Exams are different in this course; I find that they are less necessary
  - Post HW solutions after deadline
    - Doesn't really work with unlimited extensions ☹
  - Extra Credit
    - Haven't really thought about it much, instead using a "mastery" grading approach. Will think if there are ways to integrate this.

# Lecture Outline

- **The OS**
- POSIX I/O
- Locality

# Remember This?

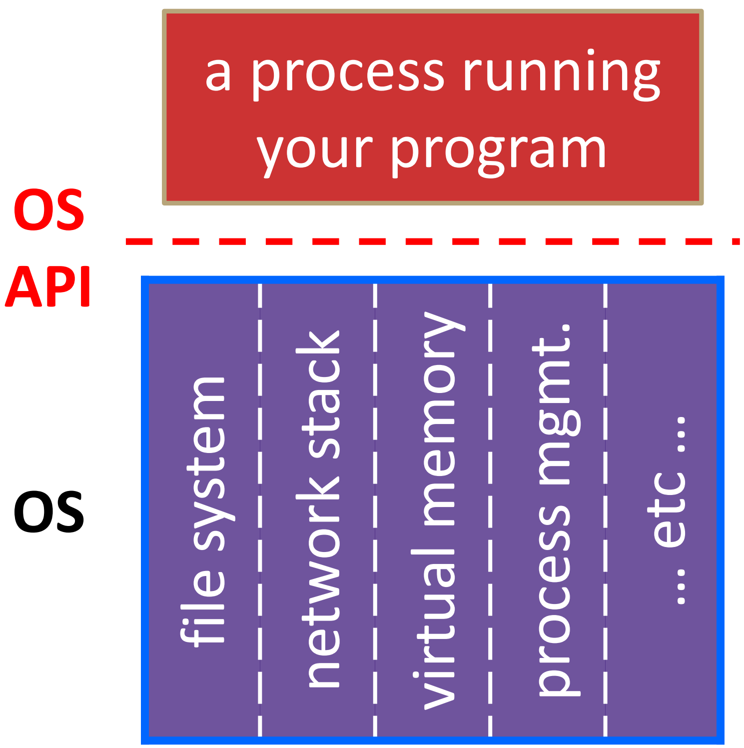


# What's an OS?

- Software that:
  - Directly interacts with the hardware
    - OS is trusted to do so; user-level programs are not
    - OS must be ported to new hardware; user-level programs are portable
  - Abstracts away messy hardware devices
    - Provides high-level, convenient, portable abstractions (*e.g.* files, disk blocks)
  - Manages (allocates, schedules, protects) hardware resources
    - Decides which programs have permission to access which files, memory locations, pixels on the screen, etc. and when

# OS: Abstraction Provider

- The OS is the "layer below"
  - A module that your program can call (with system calls)
  - Provides a powerful OS API – POSIX, Windows, etc.




**File System**
- open(), read(), write(), close(), …

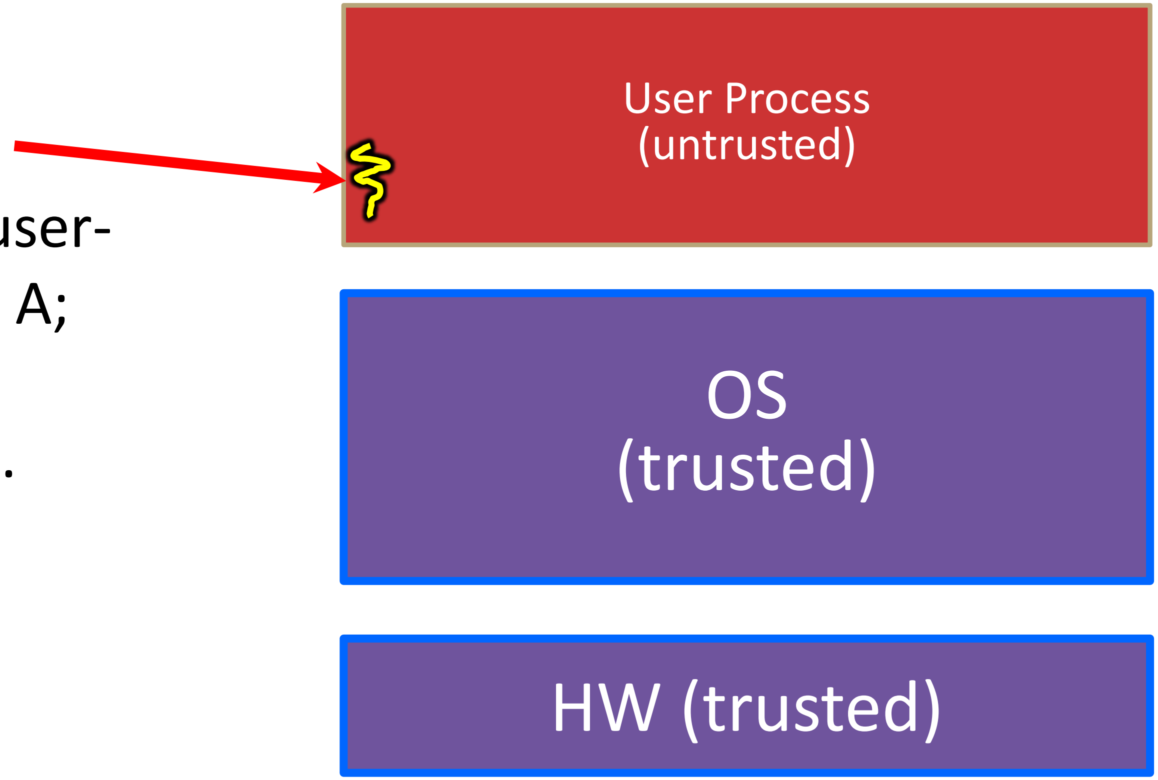
**Network Stack**
- connect(), listen(), read(), write(), ...

**Virtual Memory**
- brk(), shm_open(), …

**Process Management**
- fork(), wait(), nice(), …

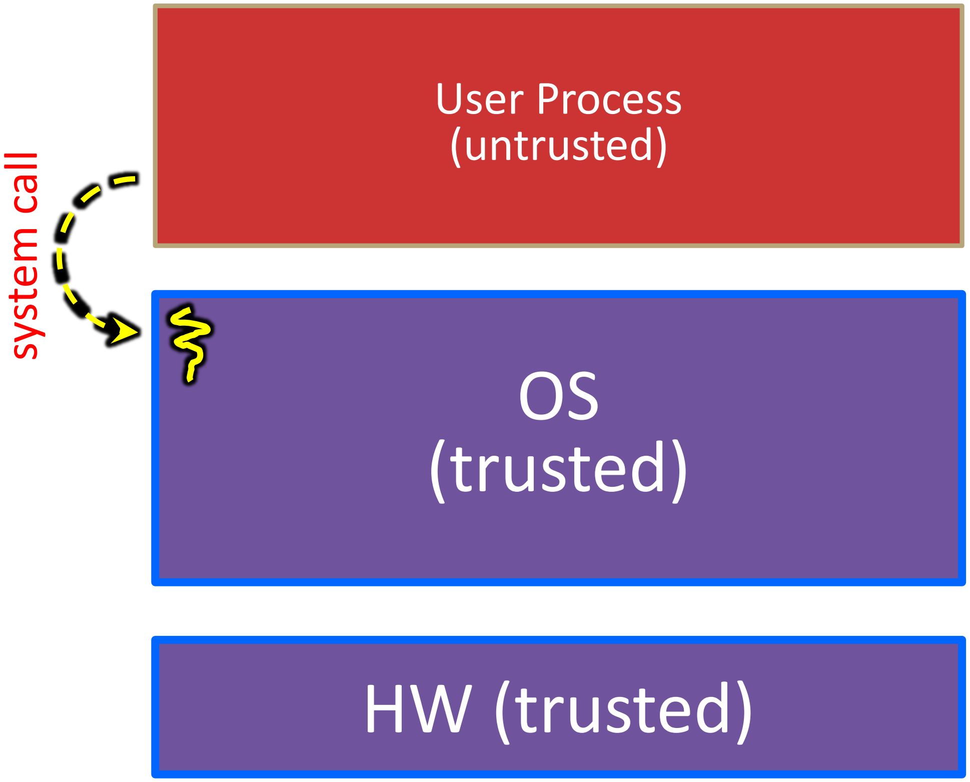
# System Call Trace (high-level view)

A CPU (thread of execution) is running user-level code in Process A; the CPU is set to *unprivileged mode*.



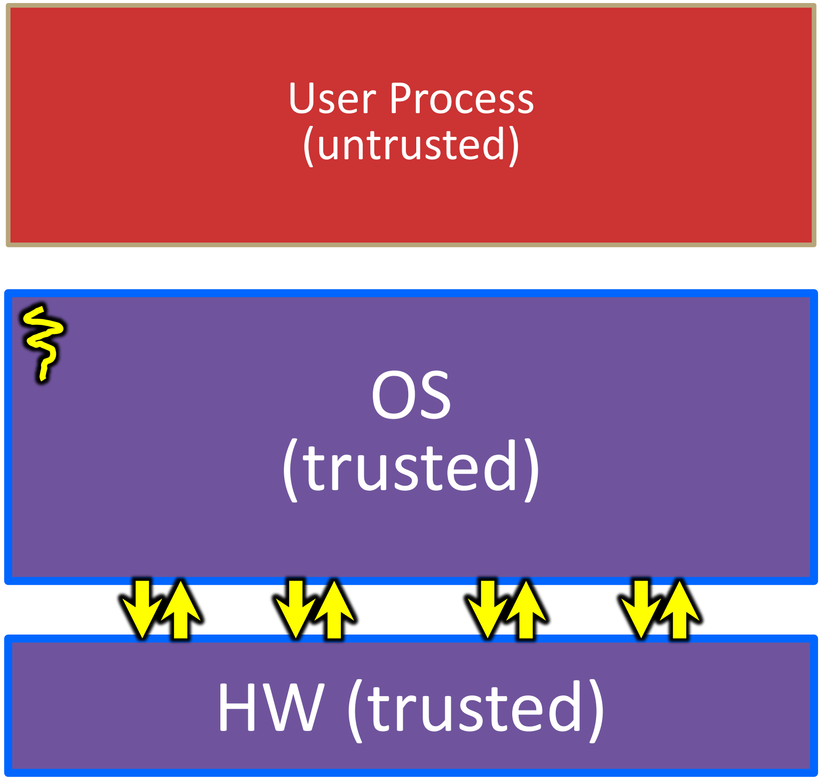
# System Call Trace (high-level view)

Code in Process invokes a system call; the hardware then sets the CPU to *privileged mode* and traps into the OS, which invokes the appropriate system call handler.



# System Call Trace (high-level view)

Because the CPU executing the thread that's in the OS is in privileged mode, it is able to use *privileged instructions* that interact directly with hardware devices like disks.

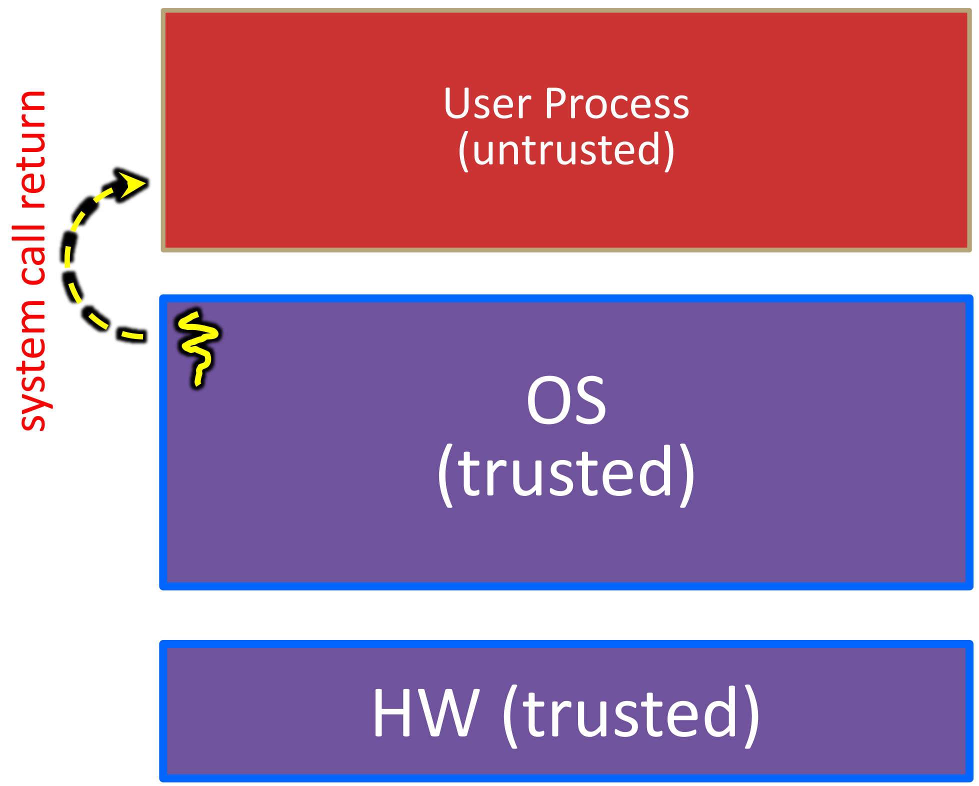


# System Call Trace (high-level view)

Once the OS has finished servicing the system call, which might involve long waits as it interacts with HW, it:

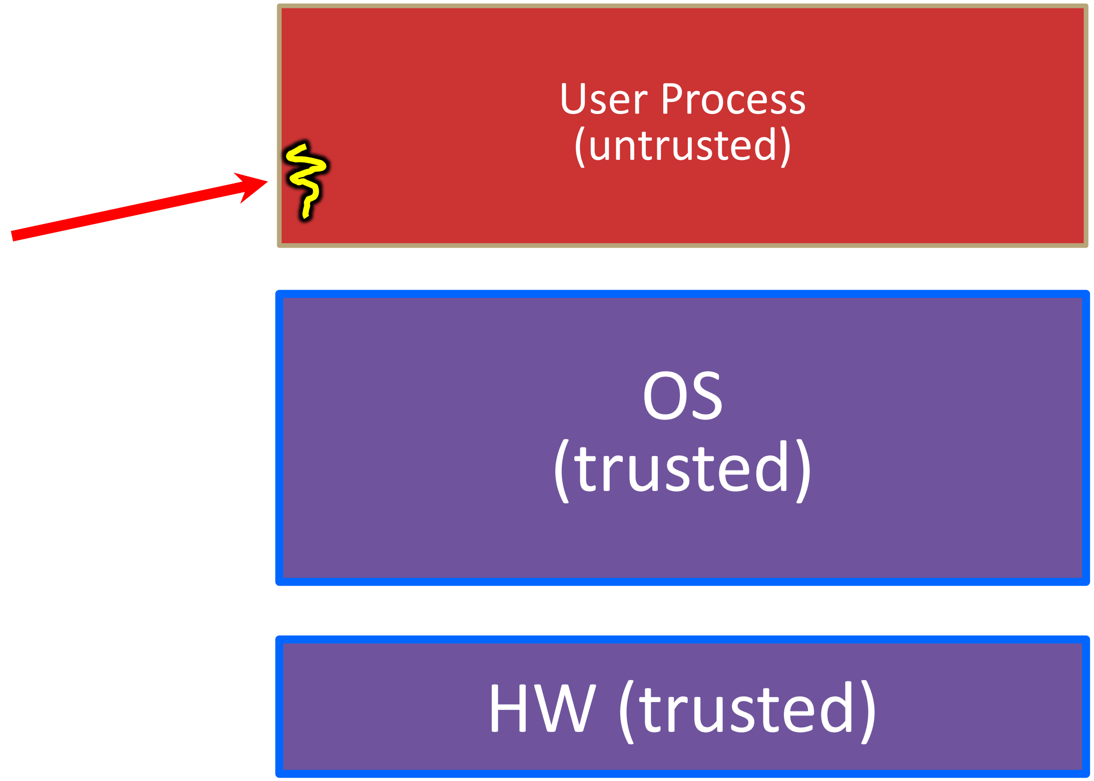
(1) Sets the CPU back to unprivileged mode and

(2) Returns out of the system call back to the user-level code in Process A.



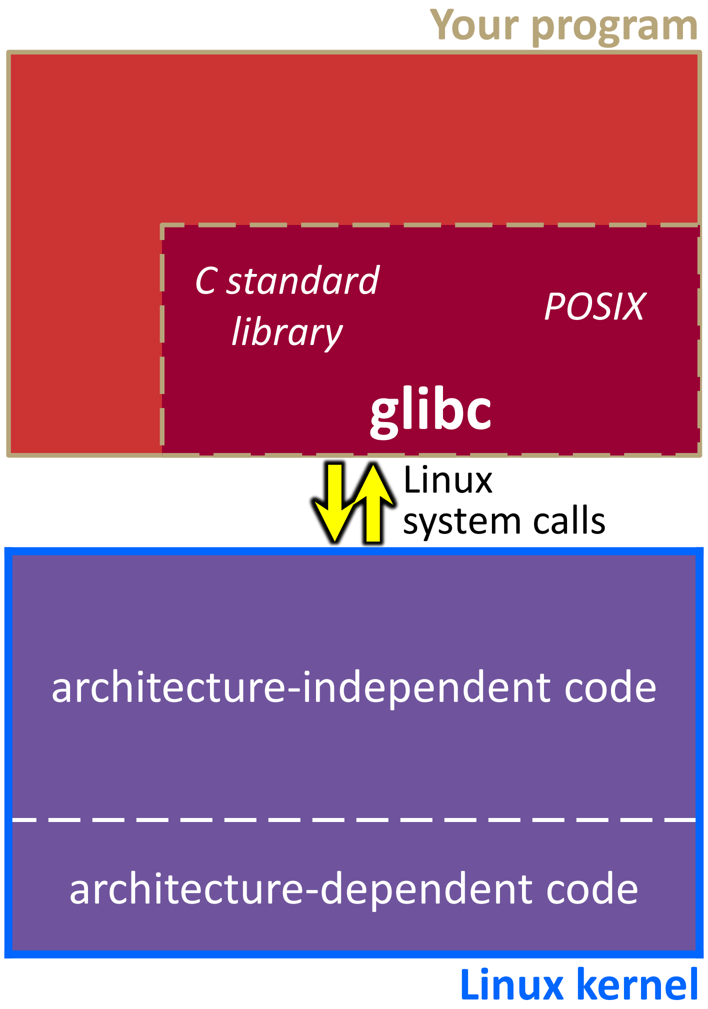
# System Call Trace (high-level view)

The process continues executing whatever code is next after the system call invocation.



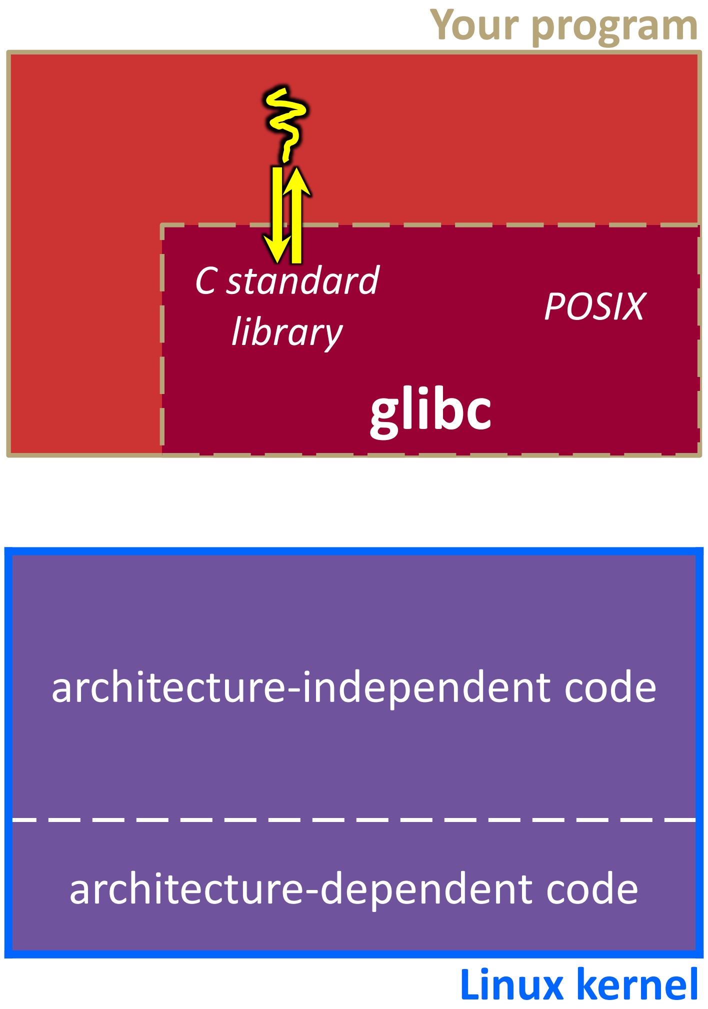
# "Library calls" on x86/Linux

- A more accurate picture:
  - Consider a typical Linux process
  - Its thread of execution can be in one of several places:
    - In your program's code
    - In `glibc`, a shared library containing the C standard library, POSIX, support, and more
    - In the Linux architecture-independent code
    - In Linux x86-64 code



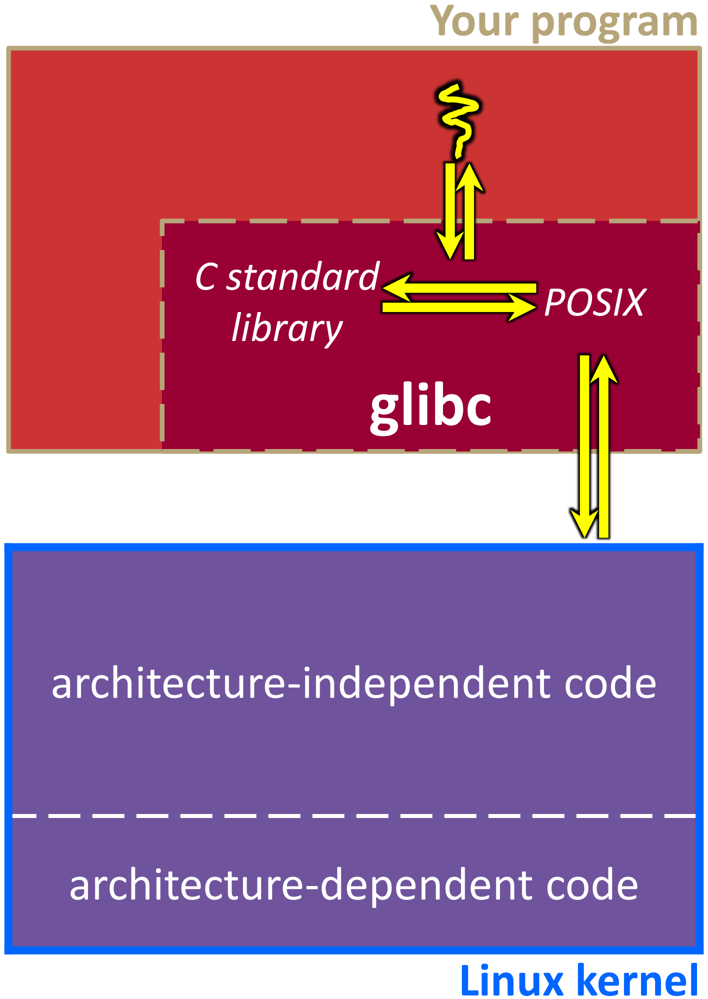
# "Library calls" on x86/Linux: Option 1

- Some routines your program invokes may be entirely handled by `glibc` without involving the kernel
  - *e.g.* **`strcmp`**`()` from `stdio.h`
  - There is some initial overhead when invoking functions in dynamically linked libraries (during loading)
    - But after symbols are resolved, invoking `glibc` routines is basically as fast as a function call within your program itself!



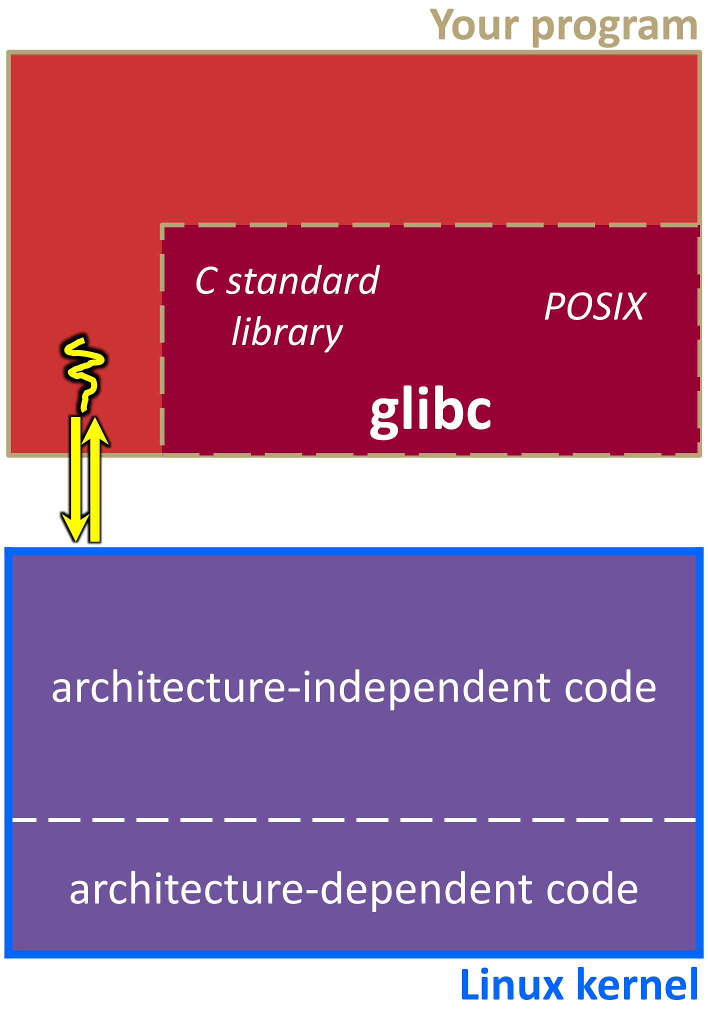
# "Library calls" on x86/Linux: Option 2

- Some routines may be handled by `glibc`, but they in turn invoke Linux system calls
  - *e.g.* POSIX wrappers around Linux `syscalls`
    - POSIX **`readdir`**`()` invokes the underlying Linux **`readdir`**`()`
  - *e.g.* C `stdio` functions that read and write from files
    - **`fopen`**`()`, **`fclose`**`()`, **`fprintf`**`()` invoke underlying Linux **`open`**`()`, **`close`**`()`, **`write`**`()`, etc.



# "Library calls" on x86/Linux: Option 3

- Your program can choose to directly invoke Linux system calls as well
  - Nothing is forcing you to link with `glibc` and use it
  - But relying on directly-invoked Linux system calls may make your program less portable across UNIX varieties



# A System Call Analogy

- The OS is a very wise and knowledgeable wizard
  - It has many dangerous and powerful artifacts, but it doesn't trust others to use them. Will perform tasks on request.
- If a civilian wants to access a "magical" feature, they must fill out a request to the wizard.
  - It takes some time for the wizard to start processing the request, they must ensure they do everything safely
  - The wizard will handle the powerful artifacts themselves. The user WILL NOT TOUCH ANYTHING.
  - Wizard will take a second to analyze results and put away artifacts before giving results back to the user.

# If You're Curious

- Download the Linux kernel source code
  - Available from http://www.kernel.org/
- `man`, section 2: Linux system calls
  - `man 2 intro`
  - `man 2 syscalls`
- `man`, section 3: `glibc`/`libc` library functions
  - `man 3 intro`
- *The* book: *The Linux Programming Interface* by Michael Kerrisk (keeper of the Linux man pages)

# Lecture Outline

- The OS
- **POSIX I/O**
- Locality

# Aside: File I/O & Disk

- File System:
  - Provides long term storage of data:
    - persists after a program terminates
    - persists after computer turns off
  - Data is organized into files & directories
    - A directory is pretty much a “Folder”
  - Interaction with the file system is handled by the operating system and hardware

# C Standard Library I/O

- In 5930, you've seen the C standard library to access files
  - Use a provided `FILE*` *stream* abstraction
  - **`fopen`**`()`, **`fread`**`()`, **`fwrite`**`()`, **`fclose`**`()`, **`fseek`**`()`
- These are convenient and portable
  - They are buffered*
  - They are implemented using lower-level OS calls

# From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface
  - `open()`, `read()`, `write()`, `close()`, `lseek()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient
  - C stdlib doesn't provide everything POSIX does
    - You will have to use these to read file system directories and for network I/O, so we might as well learn them now

# open()/close()

- To open a file:
  - Pass in the filename and access mode
    - Similar to fopen()
  - Get back a "file descriptor"
    - Similar to FILE* from fopen(), but is just an int
    - Defaults: 0 is stdin, 1 is stdout, 2 is stderr
      - -1 indicates error

Used to identify a file w/ the OS

```
#include <fcntl.h>    // for open()
#include <unistd.h>   // for close()
  ...
  int fd = open("foo.txt", O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...
  close(fd);
```

# Reading from a File

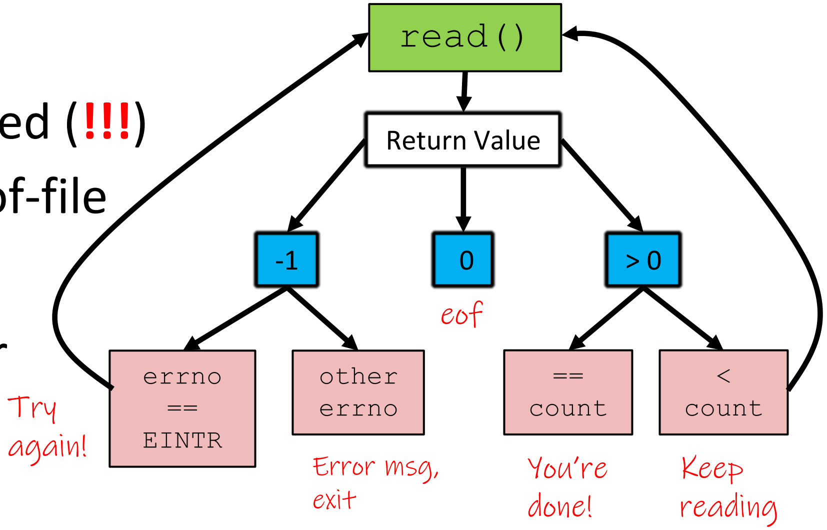
Stores read result in buf

Number of bytes

```
ssize_t read(int fd, void* buf, size_t count);
```

signed

- Returns the number of bytes read
  - Might be fewer bytes than you requested (!!!)
  - Returns **0** if you're already at the end-of-file
  - Returns **-1** on error (and sets `errno`)
  - Advances forward in the file by number of bytes read




- There are some surprising error modes (check `errno`)
  - `EBADF`: bad file descriptor
  - `EFAULT`: output buffer is not a valid address
  - `EINTR`: read was interrupted, please try again (ARGH!!!! 😤😠)
  - And many others…

Defined in errno.h

# Poll Everywhere

pollev.com/tqm

- Let's say we want to read 'n' bytes. Which is the correct completion of the blank below?

```
char* buf = ...;   // buffer of size n
int bytes_left = n;
int result;          // result of read()

while (bytes_left > 0) {
  result = read(fd, ______, bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened,
      // so return an error result
    }
    // EINTR happened,
    // so do nothing and try again
    continue; Keyword that jumps
              to beginning of loop
  }
  bytes_left -= result;
}
```

A. buf

B. buf + bytes_left

C. buf + bytes_left - n

D. buf + n - bytes_left

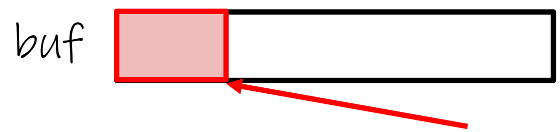
E. We're lost…

# Poll Everywhere

pollev.com/tqm

- Let's say we want to read 'n' bytes. Which is the correct completion of the blank below?

if first read only reads n/4 bytes




Want to start reading here
buf + n/4

bytes_left = n * 3/4

= buf + n - bytes_left

```
char* buf = ...;  // buffer of size n
int bytes_left = n;
int result;       // result of read()

while (bytes_left > 0) {
  result = read(fd, _______, bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened,
      // so return an error result
    }
    // EINTR happened,
    // so do nothing and try again
    continue;
  }
  bytes_left -= result;
}
```

Keyword that jumps to beginning of loop

A. buf

B. buf + bytes_left

C. buf + bytes_left - n

D. buf + n - bytes_left

E. We're lost…

# One method to `read()` $n$ bytes

```
int fd = open(filename, O_RDONLY);
char* buf = ...;  // buffer of appropriate size
int bytes_left = n;
int result;

while (bytes_left > 0) {
  result = read(fd, buf + (n - bytes_left), bytes_left);
  if (result == -1) {
    if (errno != EINTR) {
      // a real error happened, so exit the program
      // print out some error message to cerr
      exit(EXIT_FAILURE);
    }
    // EINTR happened, so do nothing and try again
    continue;  Keyword that jumps to beginning of loop
  } else if (result == 0) {
    // EOF reached, so stop reading
    break;  To prevent an infinite loop
  }
  bytes_left -= result;
}
close(fd);
```

# Other Low-Level Functions

- Read man pages to learn about:
  - `write()` – write data
    - `#include <unistd.h>`
  - `lseek()` – reposition and/or get file offset
    - `#include <unistd.h>`
  - `opendir()`, `readdir()`, `closedir()` – deal with directory listings
    - Make sure you read the section 3 version (*e.g.* `man 3 opendir`)
    - `#include <dirent.h>`

- A useful shortcut sheet (from CMU): http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf

# HW1 Overview

- In HW1, you will be implementing two file readers

- SimpleFileReader
  - A relatively simple C++ class that acts as a wrapper around POSIX

- BufferedFileReader
  - Similar to SimpleFileReader but maintains an internal buffer for improver performance <u>due to locality</u>
  - Also implements token parsing

# Lecture Outline

- The OS
- POSIX I/O
- **Locality**

# Locality

- A major factor in performance is the locality of data
  - data that is "closer" is quicker to fetch
- Have you seen this?
  - More on this when talking about memory (Jeff Dean from LADIS '09)

Numbers are out of date, but order of magnitude is same

**Numbers Everyone Should Know**

| | |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 25 ns |
| Main memory reference | 100 ns |
| Compress 1K bytes with Zippy | 3,000 ns |
| Send 2K bytes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 250,000 ns |
| Round trip within same datacenter | 500,000 ns |
| Disk seek | 10,000,000 ns |
| Read 1 MB sequentially from disk | 20,000,000 ns |
| Send packet CA->Netherlands->CA | 150,000,000 ns |

# Buffering

- By default, C `stdio` uses buffering on top of POSIX:
  - When one reads with **`fread()`**, a lot of data is copied into a buffer allocated by `stdio` inside your process' address space
  - Next time you read data, it is retrieved from the buffer
    - This avoids having to invoke a system call again
  - As some point, the buffer will be "refreshed":
    - When you process everything in the buffer (often 1024 or 4096 bytes)
  - Similar thing happens when you write to a file

# Buffering Example

Arrow signifies what will be executed next

NOTE: using fopen/fread/fclose just for example.
They will NOT be used in HW1 or in the rest of the class

```
int main(int argc, char** argv) {
  char buf[2];
→ FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```

buf

| | |
|---|---|
| | |

hi.txt (disk/OS)

| | |
|---|---|
| h | i |

# Buffering Example

Arrow signifies what will be executed next

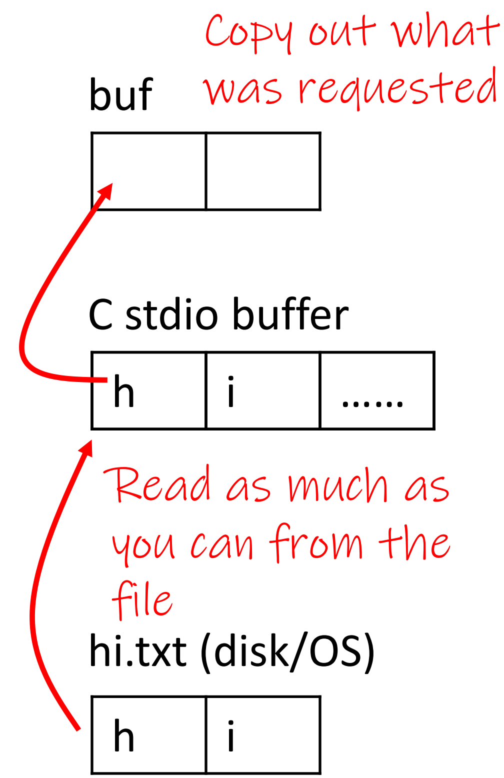
NOTE: using fopen/fread/fclose just for example.
They will NOT be used in HW1 or in the rest of the class

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```



# Buffering Example

Arrow signifies what will be executed next

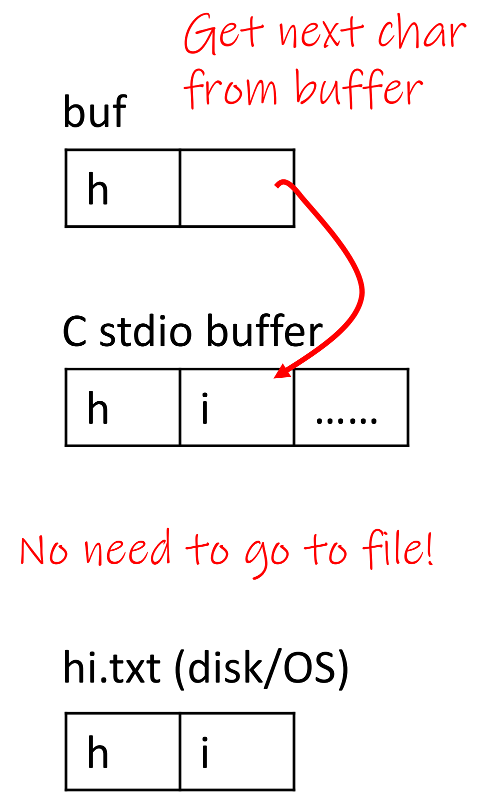
NOTE: using fopen/fread/fclose just for example.
They will NOT be used in HW1 or in the rest of the class

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);   <-- next

  fclose(fin);
  return EXIT_SUCCESS;
}
```



# Buffering Example

Arrow signifies what will be executed next

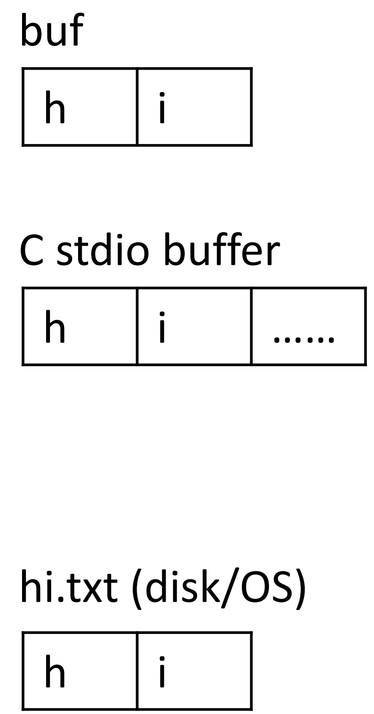
NOTE: using fopen/fread/fclose just for example.
They will NOT be used in HW1 or in the rest of the class

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

→ fclose(fin);
  return EXIT_SUCCESS;
}
```



# Buffering Example

Arrow signifies what will be executed next

NOTE: using fopen/fread/fclose just for example.
They will NOT be used in HW1 or in the rest of the class

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
→ return EXIT_SUCCESS;
}
```

buf

| | |
|---|---|
| h | i |

hi.txt (disk/OS)

| | |
|---|---|
| h | i |

# Why NOT Buffer?

- Reliability – the buffer needs to be flushed
  - Loss of computer power = loss of data
  - "Completion" of a write (*i.e.* return from `fwrite()`) does not mean the data has actually been written

- Performance – buffering takes time
  - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)

- When is buffering faster? Slower?

Many small writes
Or only writing a little

Large writes