

# Low-Level I/O – the POSIX Layer

## CSE 333 Autumn 2018

**Instructor:** Hal Perkins

**Teaching Assistants:**

Tarkan Al-Kazily

Renshu Gu

Travis McGaha

Harshita Neti

Thai Pham

Forrest Timour

Soumya Vasisht

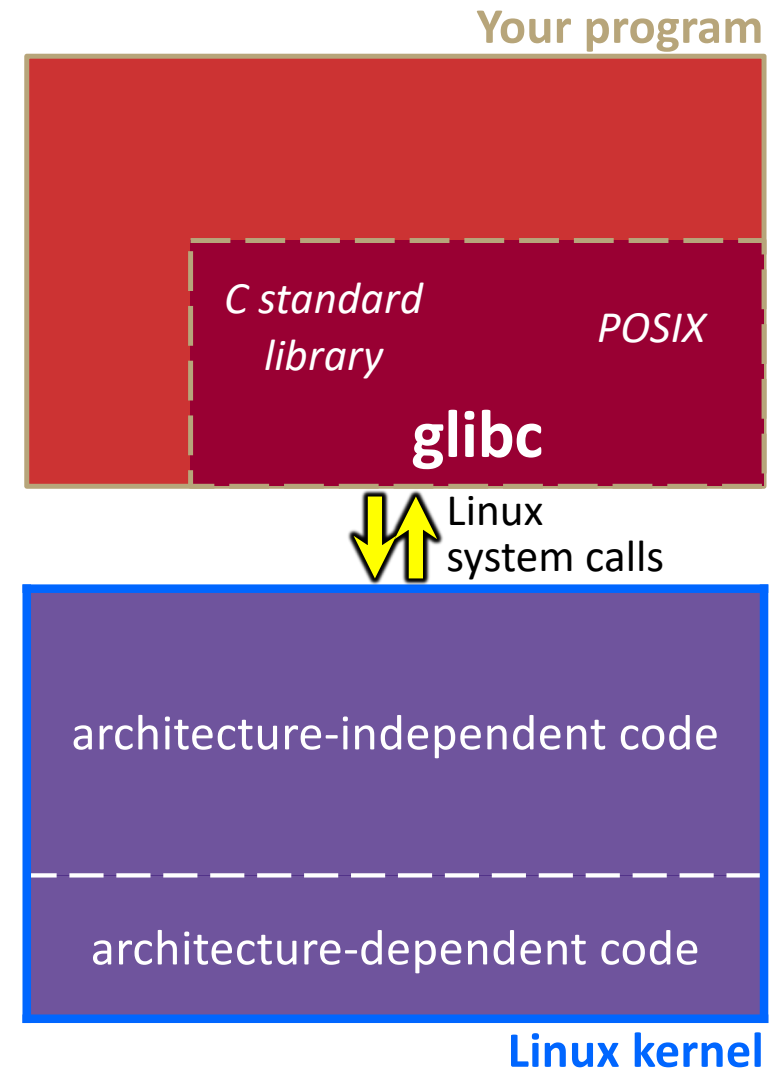
Yifan Xu

# Lecture Outline

- ❖ **POSIX Lower-Level I/O**

# Remember This Picture?

- ❖ Your program can access many layers of APIs:
  - C standard library
  - POSIX compatibility API
  - Underlying OS system calls



# C Standard Library File I/O

- ❖ So far you've used 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

# Lower-Level File Access

- ❖ 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 C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient
  - 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`

```
#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

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

- 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
- **read** has some surprising error modes...

# Read error modes

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

- On error, `read` returns -1 and sets the global `errno` variable
- You need to check `errno` to see what kind of error happened
  - `EBADF`: bad file descriptor
  - `EFAULT`: output buffer is not a valid address
  - `EINTR`: read was interrupted, please try again (ARGH!!!! 🤔😡)
  - And many others...



# One way 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 return an error result
        }
        // EINTR happened, so do nothing and try again
        continue;
    } else if (result == 0) {
        // EOF reached, so stop reading
        break;
    }
    bytes_left -= result;
}

close(fd);
```

# Other Low-Level Functions

- ❖ Read man pages to learn about:
  - `write` () – write data
  - `fsync` () – flush data to the underlying device
  - `opendir` (), `readdir` (), `closedir` () – deal with directory listings
    - Make sure you read the section 3 version (*e.g.* `man 3 opendir`)
  
- ❖ A useful cheat sheet (from CMU):  
<http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf>
  
- ❖ More in sections this week....