File I/O and The Operating System

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Poll Everywhere

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW03 Due Thursday (was due Tomorrow) *****
  - Free extension till the end of Thursday due to Rosh Hashanah
  - Please feel free to reach out if there is anything we forget, any holidays not explicitly in the university policy or anything else that comes up.

- HW04 posted tomorrow
  - Should be less work than HW03 I hope?
  - You implement some file reader objects

- Check-in Due before lecture
  - Expect the re-opens to be processed during lecture

Lecture Outline

- **C++: file streams**
- What is an OS & a System Call
- POSIX I/O

Aside: File I/O & Disk

- File System:
  - Provides long term storage of data:
    - Persist 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. (To make sure a program doesn't put the entire file system into an invalid state)

C++ fstream

- C++ gives programmers access to the file system via the `fstream` class.
- `fstream` is a high-level abstraction for accessing files, that supports formatted input/output operations.
- Supports both reading and writing
- Formatted input and output is done through `operator>>` and `operator<<` respectively.

fstream example

I am not expecting you to memorize this

```
#include <fstream>

int main() {
  fstream file("example.txt");

  int i{};
  string str, line;

  file >> str;

  if (!file) {
    cerr << "ERROR ENCOUNTERED" << endl;
  }

  file >> i;
  getline(file, line);

  file.close();
  file.open("other.md", ios_base::out | ios_base::app);

  file << "can also write to a file" << endl;
}
```

Construct an fstream open to “example.txt” for reading and writing

Reads a string. Stops at whitespace. Skips any whitespace till a string is hit.

Can treat the fstream as a bool to see if an error occurred

Reads an integer from the file!

Gets a line from the file

Close a file explicitly (destructor would have also closed it)

Re-open a file, and be explicit about which “modes” it is open in.

Write to the file

Access Mode, ifstream, ofstream

- Can pass an optional parameter to the constructor or open() to specify access type. By default gives read & write permissions:
  - `fstream file("example.txt", ios_base::in | ios_base::out);`

- Variants on fstream exist that are dedicated to just reading (ifstream) or writing (ofstream).
  - Mostly the same thing, just specific to

What is a stream?

- Any guesses? We've seen it before…
  - The concept of a stream is kinda vague.

- A "stream" is a sequence of bytes that can be accessed sequentially.
  Over time a stream *may* continue "producing" or "consuming" an unlimited number of bytes.
  - Sort of like the idea of a real-world "stream" (a continuous body of flowing water).

- Streams provide a nice interface: a sequential access of bytes. However, it may be a lot of work to maintain this abstraction/interface.

- We most commonly apply the idea of streams to files, but could be applied to the network and strings as well

Example: String Stream

```
#include <sstream>

using namespace std;

int main() {
  // extracting substrings from a string
  stringstream ss{"Hello! How are you?"};
  string token;

  while (ss >> token) {
    cout << token << endl;
  }

  // building up a string, sorta like stringbuilder
  stringstream other{};

  other << "blah blah\n";
  other << "luvsic pt.";
  other << 3;

  cout << other.str() << endl;
}
```

Include is **sstream** not stringstream

Interface behaves very similar to fstream!
Except the source is a string instead of a file

Ed Discussion

- Some programming with fstream!

Lecture Outline

- C++: file streams
- **What is an OS & a System Call**
- POSIX I/O

How is File I/O done?

- We know how programs work on hardware from 2400, right? (hopefully)

- We know how our program translates to assembly. ASM mostly involves interacting with registers and memory.
  - Files are outside of memory, how do we interact with that?

- The OS provides us a way to interact with things that are "outside" of our program.

- There are many programs running on your computer right now, and your computer does a lot more than edit memory and registers. The OS supports this.

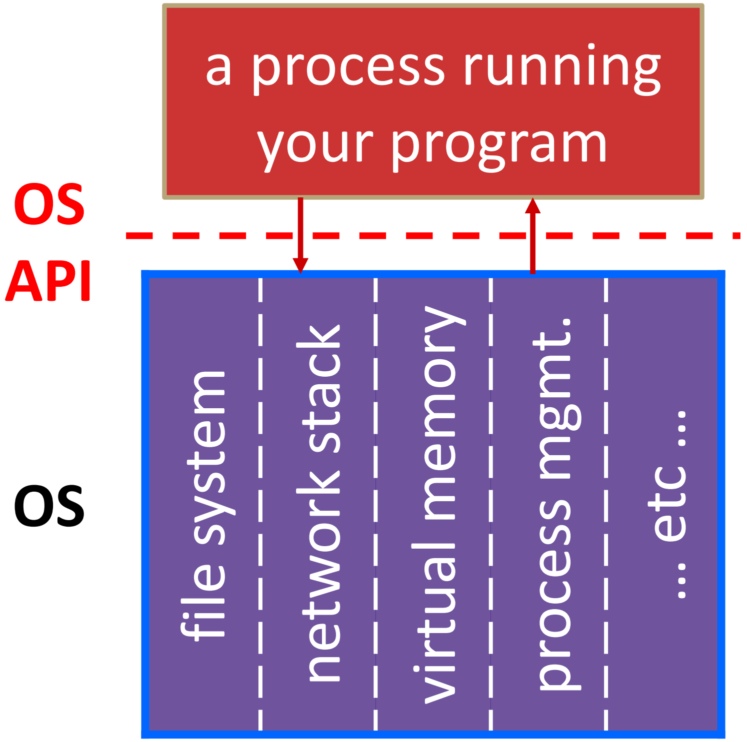
What's an OS?

- The programs we write (for the most part) are "user-level" and usually only have basic permissions. The OS has enhanced permissions.

- The OS is Software that:
  - Directly interacts with the hardware
    - OS is trusted to do so; user-level programs are not (user programs may mess it up)
    - 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.



OS provides access to many services on the computer:

- **File System**
- **Network Stack**
- **Virtual Memory**
- **Process Management**
- **Threads**
- **…**

What are system calls?

- System calls are how we invoke the functionality of the operating system.
  - Boils down to assembly
  - We put a special number in a register to specify what "function" we want the OS to run.
  - Put arguments in correct registers
  - Then run a special instruction similar to calling a function, except transfer is controlled to the OS.
    - X86/64 – `syscall`
    - ARM – `svc #0`
    - riscv – `ecall`
    - LC4 – `TRAP`
- When the OS runs, our program runs in a special "protected/higher-privilege" mode. It takes time for these abstractions to work.

How are they implemented?

- Well, we can always invoke our systemcall via in-line assembly!
  - Yes, this is legal C and C++, though your machine may use a different ASM ISA

```
int main() {
  int fd = -1;
  int mode = O_RDONLY;
  int flags = 0;
  const char* fname = "example.txt";

  __asm__("movl $2, %%eax\n"
          "movq %1, %%rdi\n"
          "movl %2, %%esi\n"
          "movl %3, %%edx\n"
          "syscall\n"
          "movl %%eax, %0"
          : "=r" (fd)
          : "r"(fname), "r"(flags), "r"(mode)
          : "rax", "rbx", "rdi", "rsi", "rdx");
```

```
movl    $-1, -16(%rbp)
movl    $0, -20(%rbp)
movl    $0, -24(%rbp)
leaq    .L.str(%rip), %rax
movq    %rax, -32(%rbp)
movq    -32(%rbp), %rcx
movl    -24(%rbp), %r8d
movl    -20(%rbp), %r9d
#APP
movl    $2, %eax
movq    %rcx, %rdi
movl    %r8d, %esi
movl    %r9d, %edx
syscall
movl    %eax, %ecx
#NO_APP
```

System Call Portability

- There are many different assembly architectures. Whenever someone wants to interact with the OS, do they need to write asm specific to that architecture?

- In the past: yes!
  - In the past (1970s and prior) ASM was how you programmed many things.
    C was considered a "high level" language.

- In the present: no!
  - To help make an OS more portable, there is a nice C level interface available to user level programs that serves as a very thin wrapper around the assembly: POSIX API

Lecture Outline

- C++: file streams
- What is an OS & a System Call
- **POSIX I/O**

From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – **P**ortable **O**perating **S**ystem **I**nterface (for Uni**x**)
  - `open()`, `read()`, `write()`, `close()`, `lseek()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their C and C++ std lib counterparts
    - Also less convenient
    - Better than using assembly!
  - C and 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

C++ Standard Library I/O

- We've seen the C++ standard library to access files
  - Uses a *stream* abstraction
  - `fstream`, `ifstream`, `ofstream`, `getline()`, etc.

- These are convenient and portable
  - They are buffered*
  - They are implemented using lower-level OS calls

ALL FILE I/O IS BUILT ON TOP OF LOWER-LEVEL OS CALLS

open()/close()

- To open a file:
  - Pass in the filename and access mode
  - Get back a "file descriptor"
    - Similar to `FILE*` from `fopen()`, but is just an $\boxed{\texttt{int}}$
      - Returns `-1` to indicate error
    - Must manually close file when done ☹

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

  - Function is written in C: follows C design
    - Takes in a file descriptor
    - Takes in an array and length of where to store the results of the read
    - Returns number of bytes read* (see next slide)
  - **EVERY TIME we read from a file, this function is getting called somewhere**
    - Even in Java or Python
    - There are wrappers around this, but they are all implemented on top of these system calls
    - The OS doesn't speak java or python, it "speaks" assembly and C so all languages must have a way to invoke these C functions.

Reading from a File

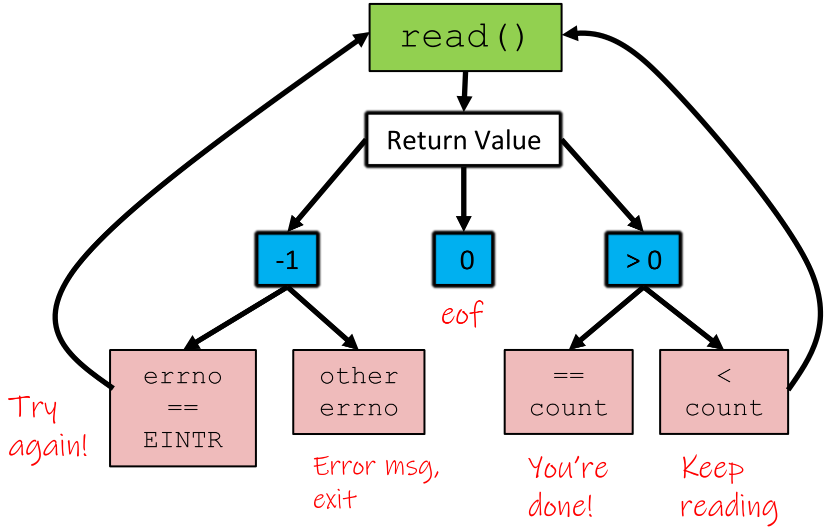
Stores read result in buf

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!!!! 😤 😠)
  - `EAGAIN`: resource temporarily unavailable, please try again
  - And many others…

Defined in `cerrno`

Example Naïve Read Code

```
int fd = open(filename, O_RDONLY);
array<char, 1024> buf {};  // buffer of appropriate size

ssize_t result = read(fd, buf.data(), 1024); // data() gets a pointer to underlying data

// No error Checking!!!!!

// If we want to construct a string from the bytes read
// we need to say how many bytes to take from the array.
string data_read(buf.data(), result);

// Whenever we are done with the file, we must close it
close(fd);
```

Ed Discussion

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

```
array<char, n> buf {}; // buffer
int bytes_left = n;
int result;         // result of read()

while (bytes_left > 0) {
  result = read(fd, ______, bytes_left);
  if (result == -1) {
    if (errno != EINTR &&
        errno != EAGAIN) {
      // 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.data()

B. buf.data() + bytes_left

C. buf.data() + bytes_left - n

D. buf.data() + n - bytes_left

E. We're lost…

Ed Discussion

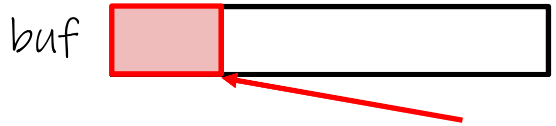
- Let's say we want to read 'n' bytes. Which is the correct completion of the blank below?

if first read only reads n/4 bytes

```
array<char, n> buf {}; // buffer
int bytes_left = n;
int result;          // result of read()

while (bytes_left > 0) {
  result = read(fd, ______, bytes_left);
  if (result == -1) {
    if (errno != EINTR &&
          errno != EAGAIN) {
      // 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



Want to start reading here
buf + n/4
bytes_left = n * 3/4
= buf + n - bytes_left

A. **buf.data()**

B. **buf.data() + bytes_left**

C. **buf.data() + bytes_left - n**

D. **buf.data() + n - bytes_left**

E. **We're lost…**

Ed Discussion

- Go to Ed and program the “wrapped_read” function ☺

One method to `read()` n bytes

```
int fd = open(filename, O_RDONLY);
array<char, 1024> buf {};  // buffer of appropriate size
int bytes_left = 1024;
int result;

while (bytes_left > 0) {
  result = read(fd, buf.data() + (1024 - bytes_left), bytes_left);
  if (result == -1) {
    if (errno != EINTR && errno != EAGAIN) {
      // 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);
```

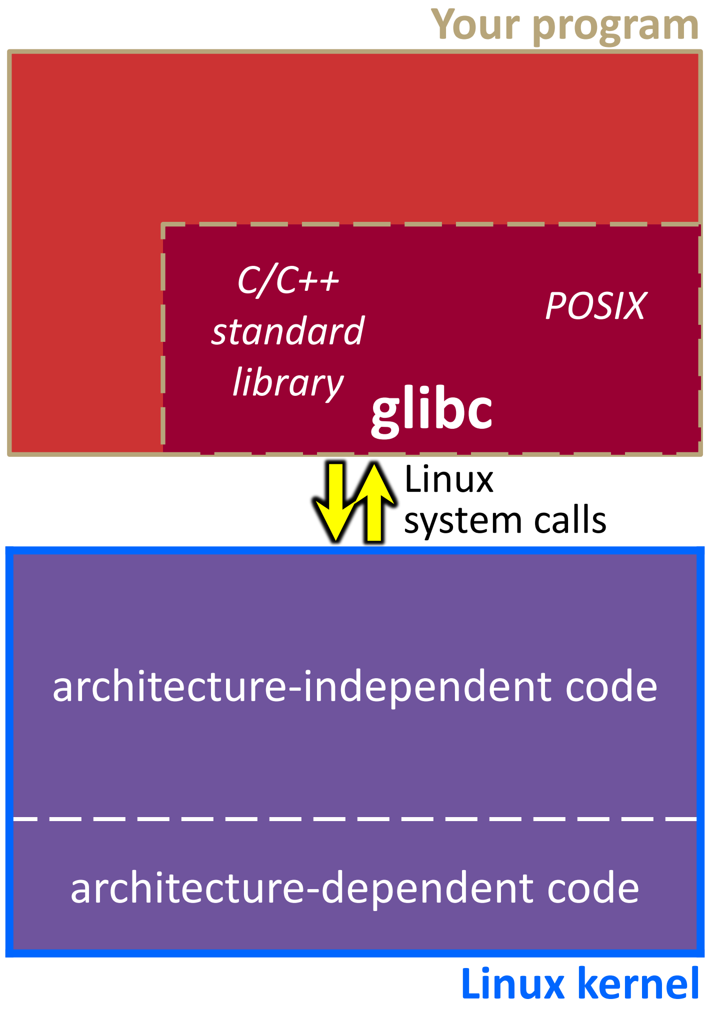
Other Low-Level Functions

- 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

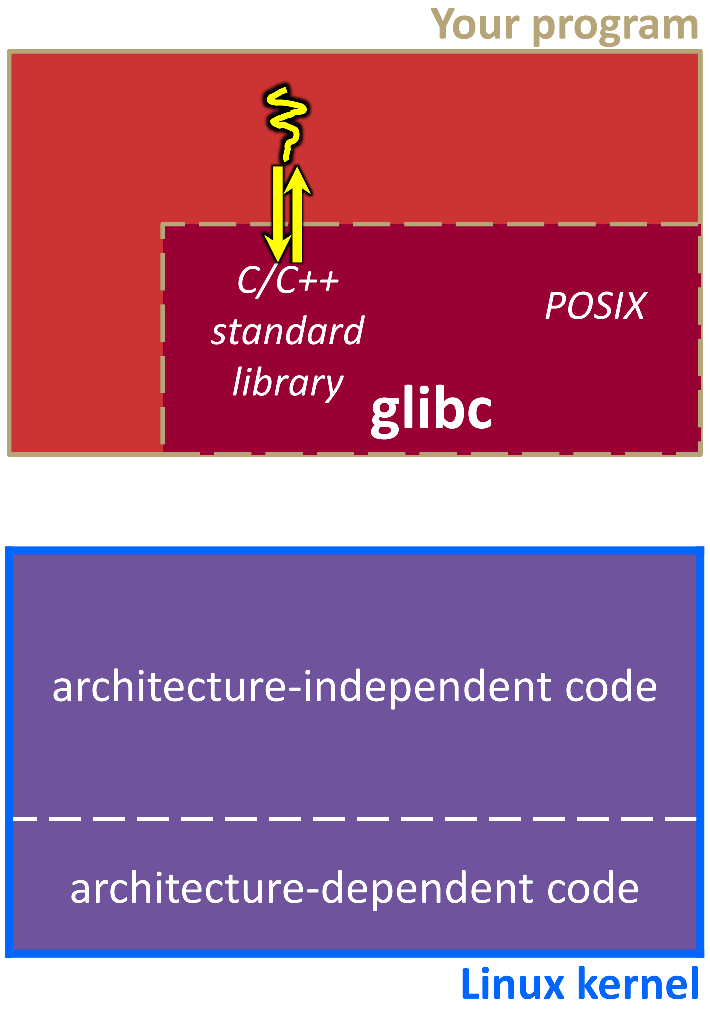
"Library calls" on x86/Linux

- A more accurate picture of what happens when we invoke a function
- The C++ standard library and POSIX are written as "user-level" code.
- The OS has both code that depends on which architecture we are on, and codes that is independent of that.



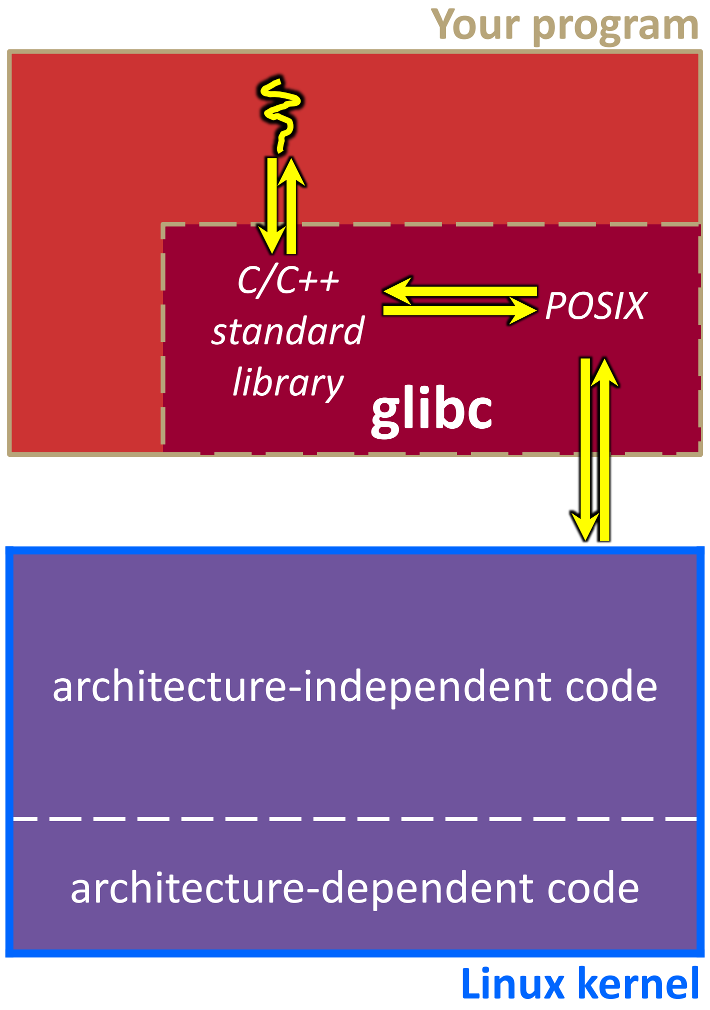
"Library calls" on x86/Linux: Option 1

- Some routines your program invokes may be entirely handled by `glibc` without involving the kernel
  - *e.g.* **`strcmp`**`()` from `stdio.h`



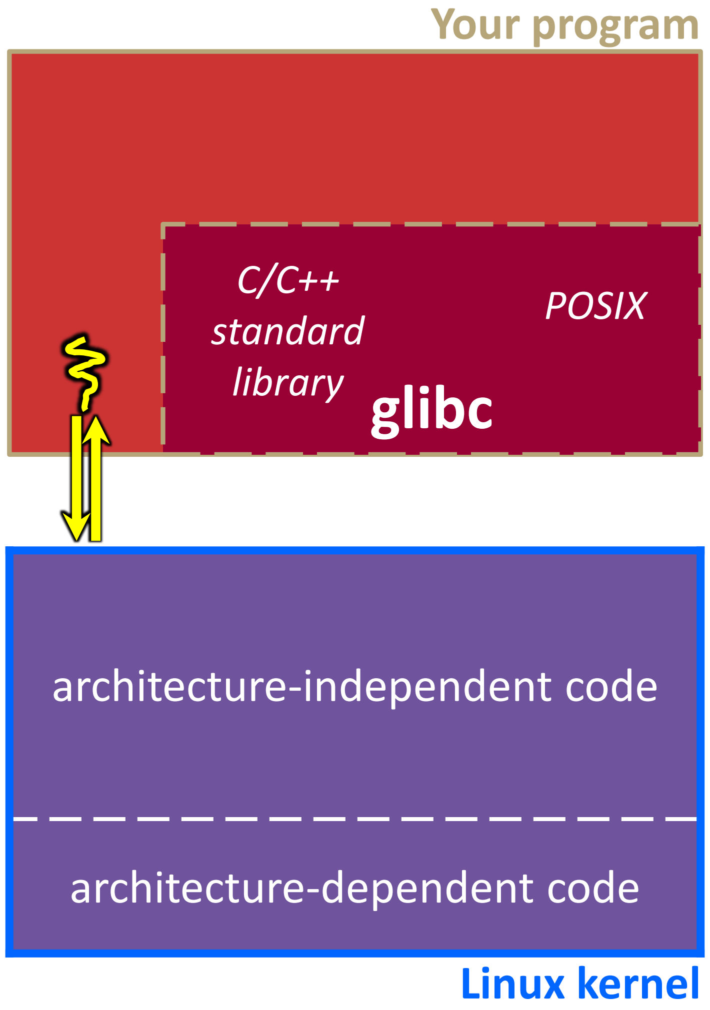
"Library calls" on x86/Linux: Option 2

- 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()`** will invoke POSIX, which invokes underlying Linux **`open()`**, **`close()`**, **`write()`**, etc.



"Library calls" on x86/Linux: Option 3

- We can also do the gross "direct" system call via assembly.
- Has its uses, but you don't want to do this 99.999999% of the time



That’s all for now!

- Next time:
  - Why would we ever use posix read when we have fstream?

- Hopefully you are doing well ☺