Low-Level I/O – the POSIX Layer

CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani	Hannah Jiang	Jen Xu
Justin Tysdal	Leanna Nguyen	Sayuj Shahi
Wei Wu	Yiqing Wang	Youssef Ben Taleb

Poll Everywhere

pollev.com/uwcse333

- Please write a 3 sentence summary of the slides after the PollEverywhere (slides 43 and beyond) from Monday's lecture ("*C File I/O & System calls*")

Administrivia (1)

- Exercises 6 and 7 both due Friday

- Today, we finish the materials for Exercise 7:
  - POSIX I/O for directories and reading data from files
  - Read a directory and open/copy text files found there
    - Copy *exactly* and ***only*** the bytes in the file(s). No extra output, no "formatting", no "titles", no other transformations.
  - Good warm-up for…

- Homework 2 due in two weeks (Tue, Oct 29)
  - File system crawler, indexer, and search engine
  - Spec available now, starter code soon!

Administrivia (2)

- Homework 1:
  - Lots of "OMG I submitted late because I forgot to allocate time for tagging" – don't do that
  - Late days are on Canvas, not on Gradescope
  - Some suggestions for using git in 333:
    - Don't checkout/branch/merge/rebase your primary repo
      - (Well, maybe to recover a previous version of a file, but only if you know how to reset the repo back to it's proper state)
    - `git pull` then checkout your tag in **a different copy** of your repo. Don't do that in your main copy!

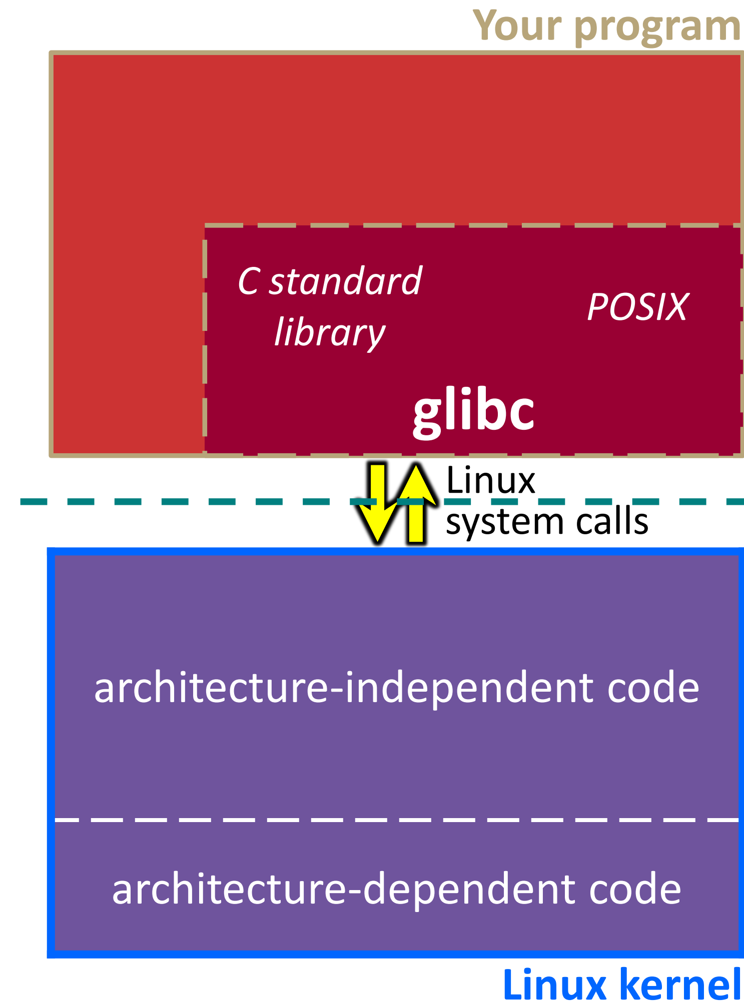
- Exercises:
  - Please remember that linter errors are **correctness errors** and therefore are docked points (this includes copyrights!)

Lecture Outline

- System Calls
- **POSIX Lower-Level I/O**

Remember This Picture?

- Your program can access many layers of APIs:
  - C standard library
    - Some are just ordinary functions (<string.h>, for example)
    - Some also call OS-level (POSIX) functions (<stdio.h>, for example)
  - POSIX compatibility API
    - C-language interface to OS system calls (fork(), read(), etc.)
  - Underlying OS system calls
    - Assembly language ☺



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
  - We 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…

Poll Everywhere

pollev.com/uwcse333

- Assume you want to read *n* bytes from a file. Which is the correct completion of the blank below?

```
char* buf = ...;  // at least 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;
}
```

add to write the result

A. buf

B. buf + bytes_left

C. buf + bytes_left - n

D. buf + n - bytes_left

E. We're lost…

One way to `read()` n bytes

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

readN.c

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

- More in sections this week…. (as in, *tomorrow!*)