File I/O: C vs POSIX

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

pollev.com/cse333

About how long did Exercise 5 take?

A. 0-1 Hours
B. 1-2 Hours
C. 3-4 Hours
D. 4+ Hours
E. I prefer not to say

Administrivia (1 of 2)

- HW 1 due tomorrow (10/10) @ 9pm
  - … as if you didn't know this already!
  - Please leave "STEP #" markers for graders!
  - Remember to tag `hw1-final` (we'll figure out late days)

- Want to request an exercise regrade? Gradescope.
- Want to request a homework regrade? Piazza/email.
  - Run `git pull` to see feedback

- *No exercise due Friday!* Exercise 6 will be released on Friday (10/11) and due the following Monday (10/14)
  - Will try to release exercises earlier in the day ← *anon. f/b!* ☺

Administrivia (2 of 2)

- Reminder: your device distracts other students!
  - If you're using a laptop, please move to back of the room ← *also anon. f/b!* ☹
- Section AD (11:30) in MGH 231 henceforth! Woohoo!

Lecture Outline

- **File I/O with the C standard library**
- File I/O with the POSIX library
- Difference: Working with Directories

File I/O with the C standard library

- We'll start by using C's standard library
  - These functions are part of `glibc` on Linux
  - They are implemented using Linux system calls (POSIX)

- C's `stdio` defines the notion of a stream
  - A sequence of characters *to* and *from* a device
    - Can be either *text* or *binary*; Linux does not distinguish
  - Is *buffered* by default; `libc` reads ahead of your program
  - Three streams provided by default: `stdin`, `stdout`, `stderr`
    - You can open additional streams to read and write to files
  - C streams are manipulated with a `FILE*` pointer, which is defined in `stdio.h`

console console unbuffered console

C Stream Functions (1 of 2)

- Some stream functions (complete list in `stdio.h`):

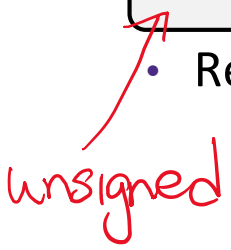
opaque to caller, like HW1!

"w", "r", etc

- `FILE* fopen(filename, mode);`
  - Opens a stream to the specified file in specified file access mode
- `int fclose(stream);`
  - Closes the specified stream (and file)
- `int fprintf(stream, format, ...);`
  - Writes a formatted C string
    - `printf(...);` is equivalent to `fprintf(stdout, ...);`
- `int fscanf(stream, format, ...);`
  - Reads data and stores data matching the format string

C Stream Functions (2 of 2)

- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)
  - `size_t fwrite(ptr, size, count, stream);`
    - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*
  - `size_t fread(ptr, size, count, stream);`
    - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*



C Streams: Error Checking/Handling

- Some error functions (complete list in `stdio.h`):

essentially a global var! so check it ASAP!

  - `void perror(message);`
    - Prints `message` followed by error message related to `errno` to `stderr`
  - `int ferror(stream);`
    - Checks if the error indicator associated with the specified stream is set, returning 1 if so
  - `int clearerr(stream);`
    - Resets error and eof indicators for the specified stream

C Streams Example

cp_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define READBUFSIZE 128

int main(int argc, char** argv) {
  FILE *fin, *fout;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;        // defined in stdlib.h
  }

  // Open the input file
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }
  ...
```

C Streams Example

If input file was 300 bytes, loop will run 4 times:

cp_example.c

```
int main(int argc, char** argv) {

  ...   // previous slide's code

  // Open the output file
  fout = fopen(argv[2], "wb");  // "wb" -> write, binary mode
  if (fout == NULL) {
    perror("fopen for write failed");
    fclose(fin);
    return EXIT_FAILURE;
  }

  // Read from the file, write to fout
  while ((readlen = fread(readbuf, 1, READBUFSIZE, fin)) > 0) {
    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
  }
  ...   // next slide's code
}
```

fout failed, but still need to free fin!

main copy loop

C Streams Example

cp_example.c

```
int main(int argc, char** argv) {

  ...   // two slides ago's code

  ...   // previous slide's code

  // Test to see if we encountered an error while reading
  if (ferror(fin)) {
    perror("fread failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

main copy loop

} the file version of free()ing your ptrs

Lecture Outline

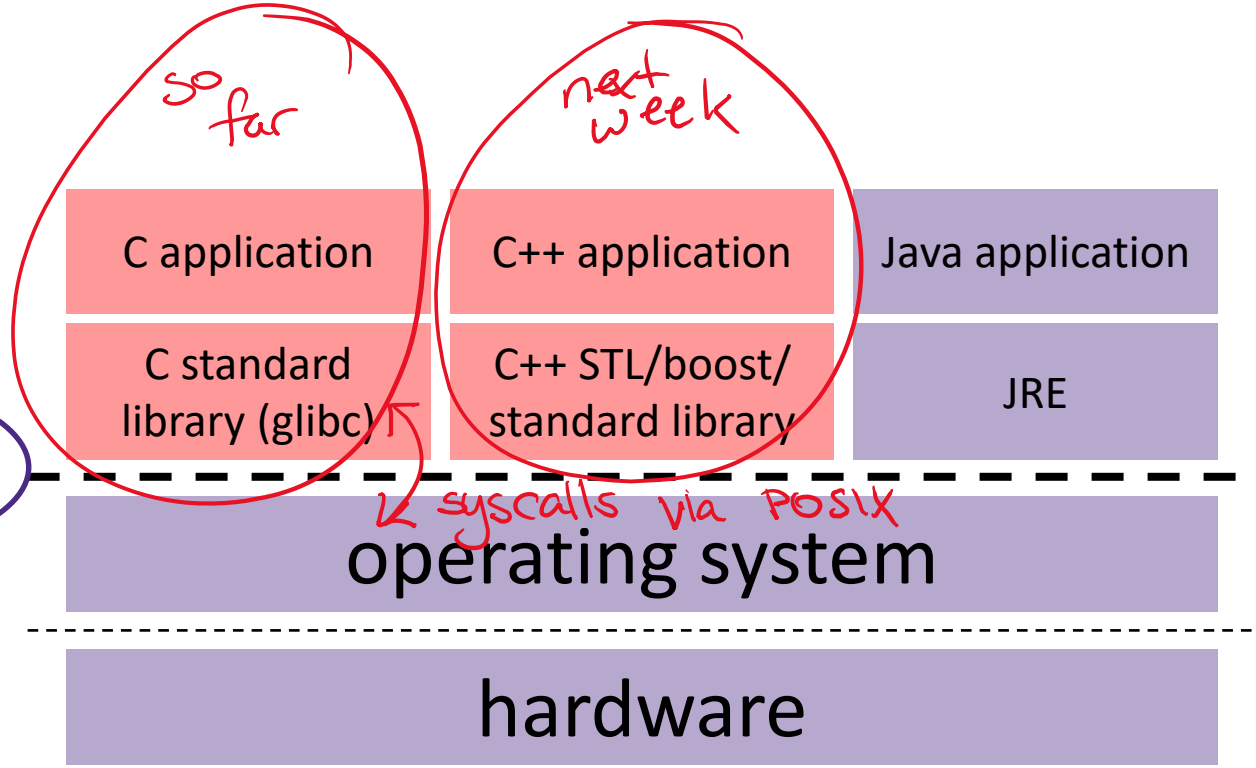
- File I/O with the C standard library
- **File I/O with the POSIX library**
- Difference: Working with Directories

Remember This Picture?

A brief diversion...

OS / app interface (system calls)

HW/SW interface (x86 + devices)



CPU memory storage network
GPU clock audio radio peripherals

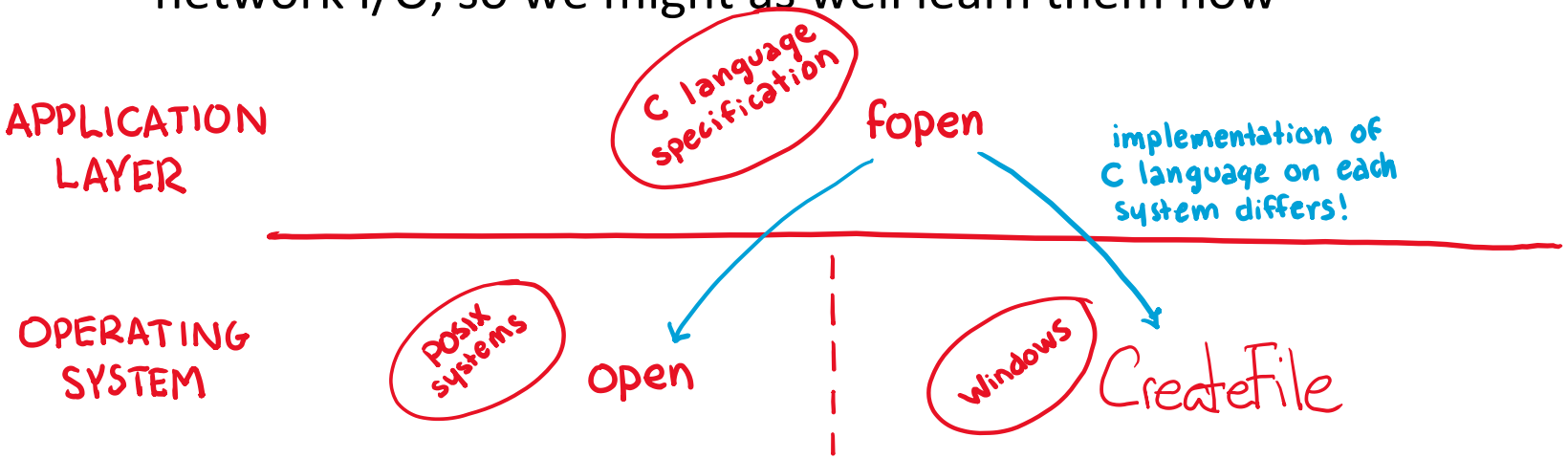
We Need To Go Deeper …

- 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

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 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

difference! Because the OS (kernel) holds the file's state instead a user-space library (i.e FILE), a ptr to a struct in kernel space would be semantically meaningless

same 3 default streams

```
#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);
```

difference! bitflags instead of strings (eg "r b")

POSIX: Reading from a File

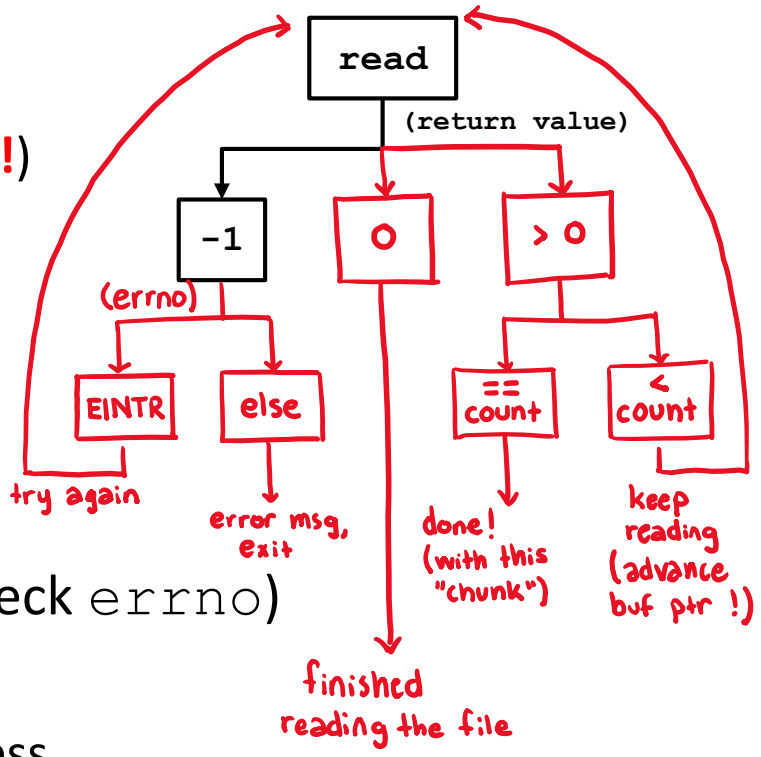
```
ssize_t read(int fd, void* buf, size_t count);
```

- 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 (and sets `errno`)
- 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…

POSIX: 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 (and sets `errno`)



- 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…

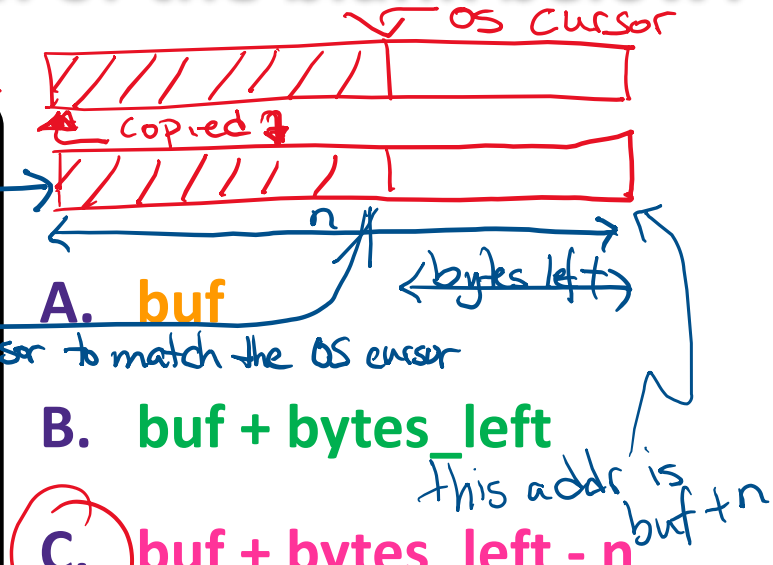
Poll Everywhere

pollev.com/cse333

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;
  }
  bytes_left -= result;
}
```



A. buf

B. buf + bytes_left

C. buf + bytes_left - n

D. buf + n - bytes_left

E. We're lost…

POSIX: 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 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 POSIX Functions

- Read man pages to learn about:
  - **`write`**`()` – write data
    - `#include <unistd.h>`
  - **`fsync`**`()` – flush data to the underlying device
    - `#include <unistd.h>`
  - **`opendir`**`()`, **`readdir`**`()`, **`closedir`**`()` – deal with directory listings ← difference! c std lib only deals w/ files
    - Make sure you read the section 3 version (*e.g.* `man 3 opendir`)
    - `#include <dirent.h>`

covered in section tomorrow

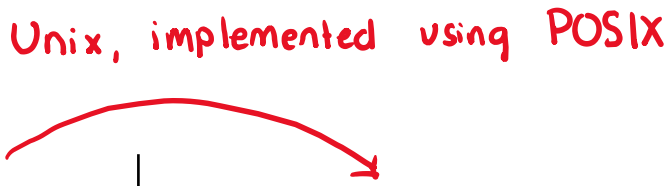
we ♡ our TAs!

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

C Standard Library vs. POSIX

	C Standard Library File I/O	POSIX I/O
Functions		
Buffering		
Implemented		
Portability		

C Standard Library vs. POSIX



	C Standard Library File I/O	POSIX I/O
Functions	fwrite, fopen	write, open
Buffering	buffered	unbuffered
Implemented	C std library	OS syscalls (*technically, a thin wrapper around them in std library)
Portability	more (works anywhere there is a C implementation)	less (specific to POSIX systems, usually Unix)

Extra Exercise #1

- Write a program that:
  - Prompts the user to input a string (use `fgets()`)
    - Assume the string is a sequence of whitespace-separated integers (*e.g.* "`5555 1234 4 5543`")
  - Converts the string into an array of integers
  - Converts an array of integers into an array of strings
    - Where each element of the string array is the binary representation of the associated integer
  - Prints out the array of strings

Extra Exercise #2

- Write a program that:
  - Loops forever; in each loop:
    - Prompt the user to input a filename
    - Reads a filename from `stdin`
    - Opens and reads the file
    - Prints its contents to `stdout` in the format shown:

```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

- Hints:
  - Use `man` to read about `fgets`
  - Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it